

AN INVESTIGATION
OF THE BOOTSTRAPPING PROCESS
AS APPLIED TO COMPILER GENERATION

Ronald Crocker Smeder

United States Naval Postgraduate School



THESIS

AN INVESTIGATION
OF THE BOOTSTRAPPING PROCESS
AS APPLIED TO COMPILER GENERATION

Ronald Crocker Smeder

Thesis Advisor:

G. A. Kildall

June 1971

Approved for public release; distribution unlimited.

T139440

LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIF. 93940

An Investigation of the Bootstrapping Process
as Applied to Compiler Generation

by

Ronald Crocker Smeder
Ensign, United States Navy
B.S., Tufts University, 1970

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1971

ABSTRACT

This paper examines the problem of compiler generation using bootstrapping techniques. It summarizes the development of several compiler systems which were produced through bootstrapping and presents the methods and formalisms through which they can be examined. The discussion of these techniques is illustrated with the development of an XPL compiler system for the Control Data 6500 computer.

TABLE OF CONTENTS

I.	Introduction.....	6
II.	XPL and the CDC 6500.....	7
	A. Why XPL?	7
	B. Characteristics of the IBM 360/67 and the CDC 6500...	8
	C. The XPL Language and Compiler System.....	9
III.	Character Sets: Establishing a Common Vocabulary.....	12
	A. Developing the Intermediate Character Set.....	12
	B. Eliminating Semantic Errors.....	13
	C. Coding and Format.....	14
IV.	The T-Diagram.	
V.	Other Compiler Implementations.....	21
	A. The NELIAC Compiler System.....	21
	B. The XPL-360/67 Compiler System.....	23
VI.	Goals for the First Compiler.....	30
	A. Resolving Conflicting Goals.....	30
	B. Goals for Compiler Generality.....	31
	C. Three Phases of Development.....	32
	D. Goals for XCOM 1.....	33
VII.	Implementation of the First Compiler.....	36
	A. Available Methods.....	36
	1. Monitor Calls.....	36

2. Library Procedures.....	37
3. Compiler Generated Procedures.....	39
B. Implementation of XCOM 1.....	40
1. Code Production and Optimization.....	40
2. Code Emission.....	44
3. The Symbol Table.....	44
C. A Monitor for the First Compiler.....	50
D. Implementation of String Variables.....	55
1. XPL 360/67 Implementation.....	55
2. Use of Bootstrapping and Library Procedures....	57
3. String Variable Structures and Operations.....	58
VIII. Conclusion.....	61
APPENDIX A: A BNF Representation of the XPL-6500 Dialect..	70
APPENDIX B: Flow Diagrams for XPL-6500 Submonitor.....	73
APPENDIX C: Program Listing of Library Procedures.....	80
APPENDIX D: Program Listing of XPL-6500 Submonitor.....	85
APPENDIX E: Program Listing of XCOM 1.....	97
List of References.....	162
Initial Distribution List.....	163
Form DD 1473.....	164

LIST OF TABLES

Table		Page
I.	IBM 360 and 6500 Character Set Comparison.....	62
II.	Description of Variable Types.....	65
III.	Major XCOM-1 Procedures.....	66
IV.	Summary of Monitor Calls.....	69

LIST OF FIGURES

Figure		
1.	Character Set Translation Algorithm.....	15
2.	360 to 6500 Bootstrapping Operation.....	17
3.	Language Processor Formalisms.....	18
4.	Evolution of First NELIAC-C Compiler.....	24
5.	Development of NELIAC-1604 Compiler.....	25
6.	Single Step Development of the NELIAC 709.....	26
7.	Evolution of XPL-360/67.....	28
8.	Structure of XPL-6500 Library.....	38
9.	XCOM-1 Code Production Procedures.....	42
10.	XCOM-1 Code Emission Procedures.....	45
11.	XCOM-1 Symbol Table Structures.....	47
12.	Symbol Table Algorithm for Local Protection.....	48
13.	Procedures Provided by SKELETON.....	49
14.	XPL-6500 Monitor Program Flow.....	51
15.	String Variable Structures.....	52

I. INTRODUCTION

The purpose of this thesis is best expressed as an attempt to examine the power and uses of the bootstrapping technique as applied to compiler generation. This investigation was conducted through the implementation of an XPL compiler for the Control Data 6500 computer.

At the outset of this project two plausible definitions of bootstrapping were made. "Evolutionary bootstrapping" was defined as the general process of building a larger system upon a subset of itself. As applied to compiler generation "bootstrapping" was defined as the process of moving a language or software system from one computer to another through the assistance of the original system.

In the process of implementing the XPL programming language upon the Control Data 6500 computer (the XPL-6500 project) both forms of bootstrapping were employed to first build the initial XPL-6500 nucleus and then to transfer it to the target computer, the CDC 6500.

This paper will first introduce the problem and systems used in the XPL-6500 project, and then discuss the various aspects of bootstrapping used to develop and implement the XPL-6500 compiler system.

II. XPL AND THE CDC 6500

The selection of XPL as the target language and the CDC 6500 as the target machine determined most of the problems to be resolved during the development of the XPL-6500 system. A brief discussion of XPL, the CDC 6500, and the reasons for their selection is provided to add insight to the problems and solutions found during the designing and implementation of XPL-6500.

A. WHY XPL?

XPL, the dialect of PL/I [Ref. 5] developed at Stanford University, was chosen for this implementation for several reasons. A primary consideration was that XPL is of sufficient complexity to require for its implementation a close examination of a wide spectrum of strategies and techniques which had been successfully used by other compiler systems. Also, XPL is a language which was specifically written for the development of compiler systems, suggesting that it would be possible to write this compiler in its own language. This would enable the compiler writer to take advantage of the existing XPL system on the IBM 360/67 for compiler development, as well as for the technique of evolutionary bootstrapping.

B. CHARACTERISTICS OF THE IBM 360/67 and the CDC 6500

The Control Data 6500 was chosen as the target machine for the new XPL system. Differences in organization between the CDC 6500 and the IBM 360/67 provided an opportunity to examine the bootstrapping process in a more general light.¹ The IBM 360/67 and the CDC 6500 are third generation computers capable of "providing extremely fast solutions to data processing, scientific, and control center problems, as well as multiprocessing, time-sharing, and management information applications." [Ref. 6, 1]

The 6500 consists of a system of eleven independent sub-computers, ten of which were designed specifically for peripheral and operating system interaction. The system 360 is normally configured as a single high speed central processing unit assisted by various channel and peripheral controllers. Information stored within the 360 system is addressable by the 32 bit word, the half-word, or the byte, which is 8 bits in size. The 6500 is addressable only to its 60 bit word, and is built upon a byte size of 6 bits. Due to this difference in addressability XPL operations involving character, or byte manipulations were found to be more difficult to implement on the CDC 6500 than on the 360/67.

Both the addressing of core and back-up storage locations in the IBM 360/67 are built upon the concepts of fixed blocks of storage. This system requires that all addressing be done relative to several registers

¹Although the CDC 6500 was used for this implementation it is understood that the discussion applies to any of the 6000 series machines.

which must be maintained by the user's program. The 6500 system also uses a method of relative addressing, but it is hidden from the user, who considers his addressing to be absolute. This difference significantly eases the implementational problems of the XPL system on the 6500 over that already existing on the 360/67 version.

In addition to these differences, the actual CDC 6500 system used at the Naval Fleet Numerical Weather Center is equipped with a one million word extended core storage (ECS) capability which proved very useful in the implementation of the first XPL system. The solutions to these organizational and operational differences will be discussed further as the actual implementation of the system is described.

C. THE XPL LANGUAGE AND COMPILER SYSTEM

The original IBM 360 version of XPL [Ref. 7] was developed basically to provide a method through which a good student language compiler could be written for the computer science students attending Stanford University. This original goal grew in size and scope until the present XPL compiler generation system developed.

The present system and language are tailored to permit the implementation of a bottom-up parsing algorithm sufficiently extended to provide table driven syntax analysis of any $(2, 1)$ bounded context grammar. The table development and syntax analysis needed by this algorithm are provided by ANALYZER, one of the four major programs of XPL compiler generation system. The analysis tables, once

produced, are used to drive the parsing algorithm included within SKELETON, the prototype compiler for the XPL system.

A basic requirement for the XPL language was that it contain all constructions necessary to implement SKELETON and ANALYZER. In this way the structure of these programs determined the basic differences of the XPL language from full PL/I.

The last two programs of this compiler system are XCOM, the XPL to machine language translator, and its submonitor, XPLSM. The submonitor, although written in assembler language, also affected the development of the XPL language. Its effect was indirect as the demands upon the language by XCOM could be changed by either including more or less of the compiler's operation within the monitor itself. The aspects of the monitor-translator interaction will be discussed again as the goals and requirements for the CDC 6500 equivalents of XCOM and XPLSM are developed.

To limit both the size and time required to develop this student language compiler, as well as improve its speed and efficiency over PL/I, only three types of variables are allowed, Fixed, Character (of variable length), and Bit. Again, to improve efficiency, XCOM was designed as a single pass compiler, thus requiring within the language that all variables be declared before being used. To facilitate the implementation of the language, the CASE statement was added to the PL/I subset. These are but a few of the many differences between

XPL and PL/I, which are documented in detail elsewhere [Ref. 8].

The BNF representation of XPL-6500 is also included in Appendix A.

It should be noted that the XPL-6500 language was developed using a 48 character set vocabulary (with composite symbols) which differs from the IBM 360 version of XPL implemented in a 60 character set vocabulary. The reasons behind this seemingly trivial alteration introduce the basic problem of finding a common vocabulary between the two systems being bootstrapped.

III. CHARACTER SETS: ESTABLISHING A COMMON VOCABULARY

The brief discussion of the general features of the IBM 360 and the CDC 6500 should indicate that the possibility of a direct linkage between the two systems would be relatively small. It was, in fact, non-existent, due mainly to the differences in "byte" sizes which then requires that any communication between the two machines be done through some mutually acceptable intermediary character set.

A. DEVELOPING THE INTERMEDIATE CHARACTER SET

Although the character sets for the IBM 360 and the CDC 6500 are about the same size, Table I shows that there are many conflicts between them. In order to minimize these conflicts, the size of the XPL character set was reduced from its present size of 60 characters to 48. Any subset properly chosen and devised would probably have proved sufficient, but in order to allow this version of the XPL language to be used on other Control Data systems smaller than the 6500, the standard PL/I 48 character set [Ref. 5] with composite symbols was selected. This 48 character set was chosen since it was both large enough to completely express the XPL language without drastically altering its syntactic structure, and yet small enough to reduce virtually all of the conflicts found between the larger character sets.

This use of a common character set which is smaller than that which expressed the language originally, requires the development of

a translation algorithm which maps the original set into this new smaller character set. The algorithm must not introduce any changes into the syntatic or semantic meaning of the original text. For a language such as XPL, there are fortunately few instances where the translated code would lose its semantic integrity through such a translation and yet retain the proper syntatic structure. These errors, which would not be detected by a parsing algorithm based entirely on the syntax of a language, must obviously be guarded against as their introduction would render the translated code unworkable and useless by filling it with semantic "bugs."

B. ELIMINATING SEMANTIC ERRORS

Two methods were employed to limit the possibilities of semantic errors due to this translation. First the reliance upon semantics hidden within the language was reduced by expanding the syntatic structure of the language. This expanded structure insures that it would then be more likely that improper syntatic constructs would occur as the result of any improper semantic interpretation during the translation process. These semantic errors would be detected as syntatic errors upon parsing of the translated text.

The second method used, was to include within the translating algorithm the ability to detect and denote semantic cases where blind translation of the original text would definitely produce semantic errors. An example of this type of error in XPL could occur if a single character

within a character string it blindly mapped into a 48 character set equivalent, which is larger in length than the original character (e.g., the mapping of the ";" into ",. ").

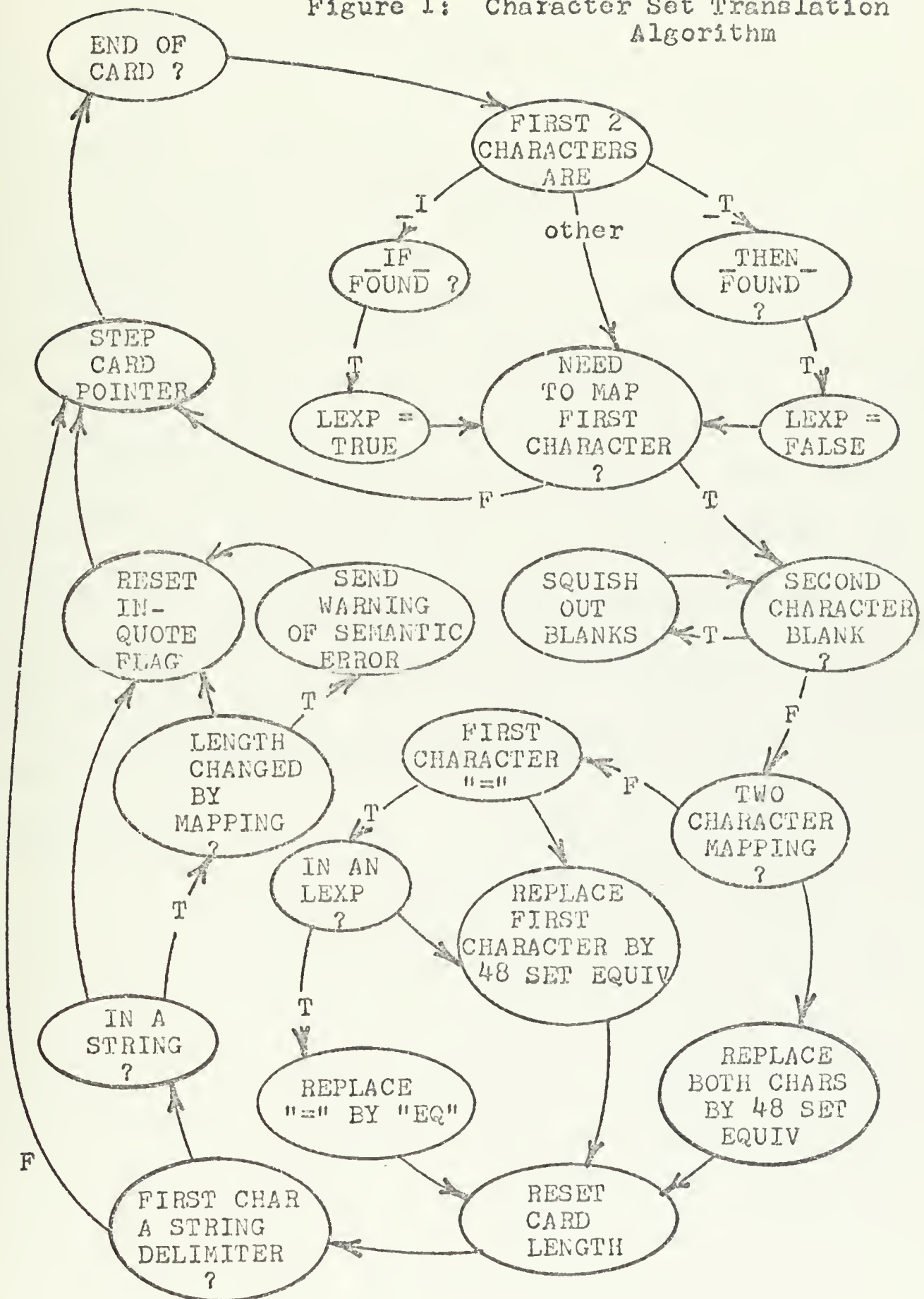
It is obvious that any translation of the original text which alters the original semantic meaning destroys the function for which the original text was designed. Figure 1 illustrates the simple algorithm used to accomplish this translation. To insure that the algorithm did not introduce any syntactic errors into the translated text, SKELETON, equipped with a 48 character set of the XPL language, verified by a syntax check that the mapped text remained syntactically error free. A test of the semantic integrity of the algorithm would not be possible until a working version of the XCOM compiler was implemented on the CDC 6500.

C. CODING AND FORMAT

Two basic characteristics of this intermediate vocabulary, coding and format, had to be acceptable to both machines. Fortunately, with the selection of the 48 character set as the intermediate vocabulary, the format and coding were easily selected.

Although not optimally efficient, the card image was chosen for the format to be used since both systems could readily accept it as a form of input, either from tape or from actual punched cards. Both methods were used in the actual implementation of the XPL system on the CDC 6500.

Figure 1: Character Set Translation Algorithm



The Hollerith code was chosen over the other available character encodings (specifically binary core image, or internal display code) as it came closest to resolving the encoding and decoding conflicts between the two machines. It became quite clear at this stage that if the translation into 48 character intermediate vocabulary had not been made, the conflicts between the character encodings required by the two machines could not have been resolved without a great deal of effort.

With this translation capability it then became possible to "map" any XPL program written for the version of XPL implemented on the IBM 360 into common vocabulary which could then be bootstrapped onto the CDC 6500. Figure 2 summarizes the implementation used to bootstrap useful XPL programs from the IBM 360 onto the CDC 6500, as well as introduce a formalism, the "T-diagram," which was found useful in both developing and expressing involved bootstrapping operations.

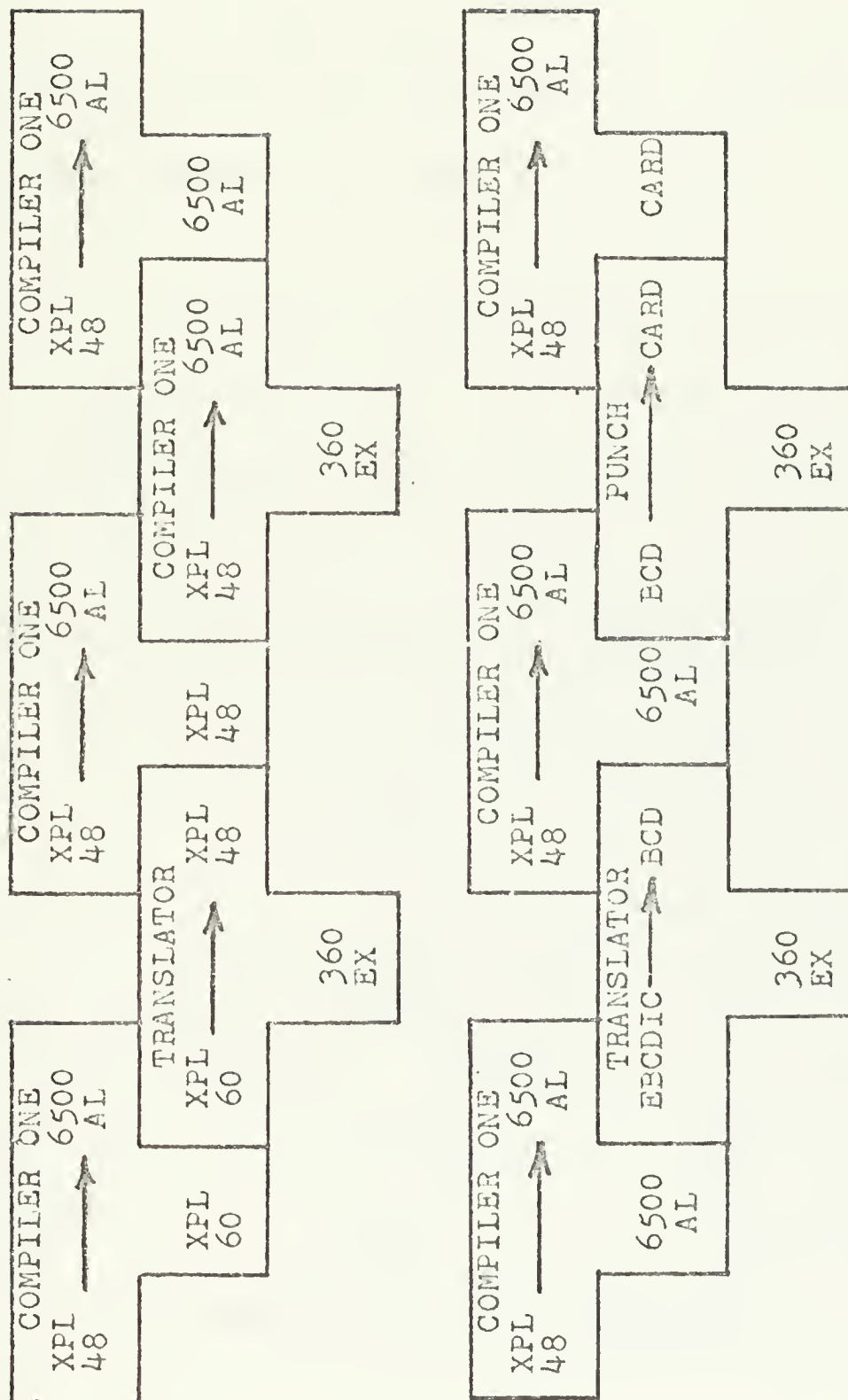


Figure 2: 360 TO 6500 BOOTSTRAPPING OPERATION

IV. THE T-DIAGRAM

The "T-Diagram" is but one of several formalisms introduced by Earley [Ref. 3] to assist in both describing the actions of processors for programming languages, and in the testing of any given set of translators and interpreters for the ability to bootstrap, or build upon themselves.

Earley's notation defines three basic classes of actions that can be performed by language processors.¹ Figure 3A represents a program written in a specific language "L1" to perform a given task "f." In Figure 3B, the T-diagram denotes a translator written in language "L1" to translate "L2" into language "L3." Finally, Figure 3C illustrates an interpreter for the language "L2" which is written in the language "L1."

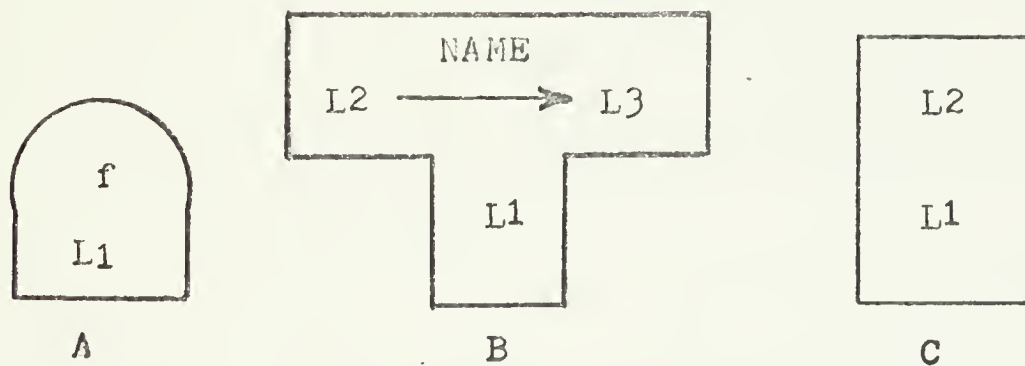


Figure 3: LANGUAGE PROCESSOR FORMALISMS

¹Figure 3 was taken from Reference 3, page 607.

Earley also presented several theorems and algorithms through which it can be determined if any given set of translators, programs, and interpreters will be able to produce a given program in a given target language. His basic premise is twofold:

1. a given set of translators, programs or interpreters could directly produce a given program in a given target language if and only if
 - a. the program already exists within the present system in some language "L2" and an interpreter for language "L2" written in the target language is present, or
 - b. the program is contained within the present set along with a compiler which will translate the program into the target language.
2. in order to cover multiple productions, he states that a given set can produce (i. e., through transitive completion) a given program in a target language if and only if there exists a sequence of productions, allowed by the first premise, which eventually leads to it.

These concepts proved to be useful not in the actual implementation of the XPL system on the 6500, but rather by providing more basic insight into the many methods which were available. Of the three formalisms defined for program, translator, and interpreter, only the "T-diagram" will be extensively used to describe the implementation

necessary to develop the XPL system for the CDC 6500. Note that this translator formalism consists of three parts. The upper section of the T-diagram applies an over-all name to the actual translation process described across its center. Although neither the name nor translation process change during the bootstrapping operation the third part of the diagram which describes the method used to implement this translation does change.

V. OTHER COMPILER IMPLEMENTATIONS

Now that a common vocabulary existed between the two machines, the development of an XPL compiler for the CDC 6500 through the use and assistance of the IBM 360 version was considered feasible. If it had been determined that a common vocabulary either could not have been created, or did not justify its attainment through sufficient gains from the XPL system, then bootstrapping of the IBM 360 version would not have been practical. The system, by necessity, would then have to have been developed using the software currently available on the CDC 6500. At this point it was found useful to examine the development of several compiler systems. Specifically, the development of the NELIAC Machine Independent Compiler System and the XPL implementation for the IBM 360 were found to be excellent examples of systems which were initially bootstrapped into existence.

A. THE NELIAC COMPILER SYSTEM

NELIAC, or Navy Electronics Laboratory International Algol Compiler, was developed concurrently with the development of the Algorithmic Language Algol 58 [Ref. 4]. Its development provides an example of the evolutionary bootstrapping technique.

The first Neliac compiler was developed for the Univac M460 computer through a bootstrapping procedure accomplished entirely within the M460 machine. This differs from the development of the

IBM and CDC version of XPL as they both used other machines and compilers to assist in the compiler implementation. The Neliac system started with a handwritten nucleus which was about twenty percent of the final compiler size. This nucleus could translate a small subset of this Algol-like language into executable code, and was used to compile newer versions which extended the scope of the original compiler. The Neliac-C compiler then developed, or evolved, by building upon previous editions of itself (i.e., evolutionary bootstrapping).

The Neliac system, once implemented through the Neliac-C compiler on the M460 computer, was used to bootstrap the Neliac system to other computers, specifically the CDC 1604, Burroughs 220, and the IBM 704. In some instances it was necessary, in order to implement an efficient version of the Neliac system, to perform a two step bootstrapping process. The first step would bootstrap an inefficient, but workable system from the old to the new computer. The second step, through the use of evolutionary bootstrapping would produce an efficient final system which was completely divorced from the original computer.

The two step bootstrapping process used in the Neliac development is exemplified in their CDC 1604 implementation. In this case the first compiler was written as an inefficient compiler to translate the Neliac into the appropriate machine language. This first step was implemented in Neliac-C on the M460 computer. This inefficient compiler was then used to recompile an improved version of itself on its target machine (in this case the 1604). In order to illustrate

this process T-diagrams are used with the following abbreviations:

406	The Univac M460.
1604	The Control Data 1604 computer.
704, 709	The IBM 704 and 709 computers.
AL	Assembly language (symbolic)
ML	Machine language (binary)
EX	Execution

Figure 4 illustrates the multi-step development of the original Neliac-C compiler on the M460 Computer. Figure 5 describes the process used to bootstrap the Neliac system from the M460 to the CDC 1604 (a two step process), and finally Figure 6 shows the single step operation needed to bootstrap the Neliac system between two similar machines such as the IBM 704 and 709.

B. XPL-360/67 COMPILER SYSTEM

As previously stated, the XPL system for the IBM 360 was developed through the assistance of another machine, the Burroughs B5500. Although the procedure used to implement the system was complex and not as direct as the bootstrapping used for the Neliac system, the amount of work expended to produce the first XPL compiler was probably far less than that spent by the Neliac originators. The developers of the XPL system chose to write two versions of their XPL to 360 machine language translator (XCOM), but to write them both in higher level languages [Ref. 7]. They chose to use Algol as

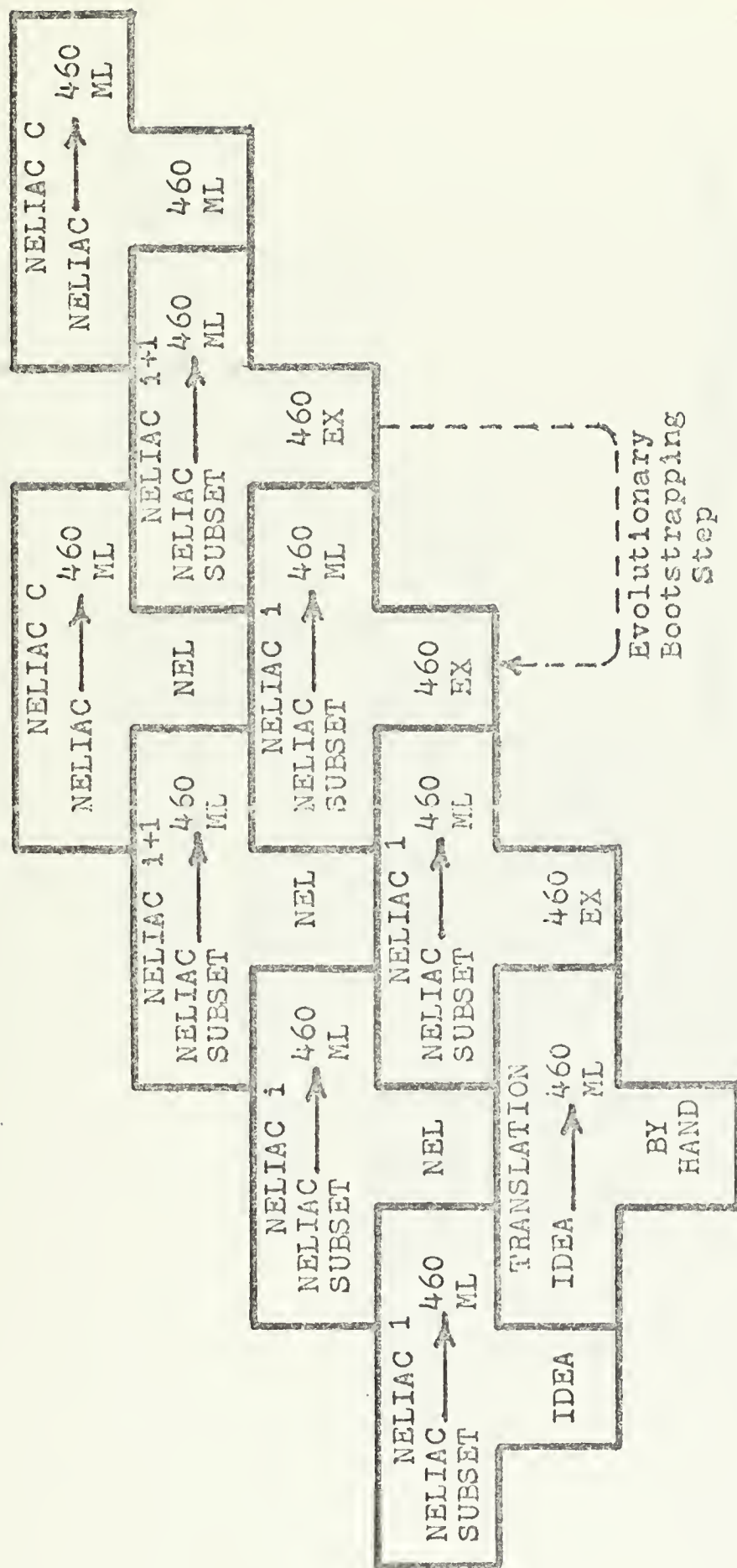


Figure 4: Evolution of First NELIAC Compiler

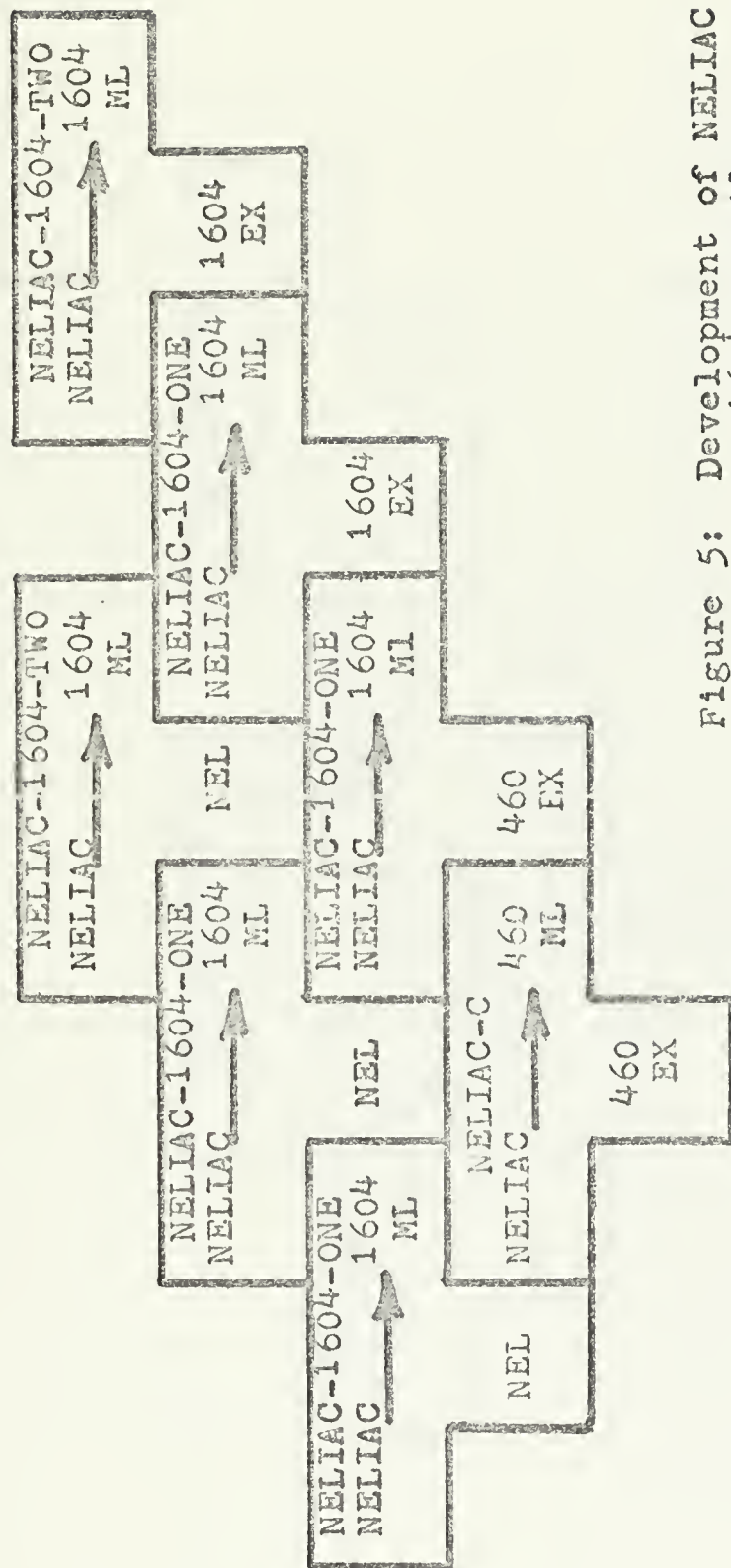
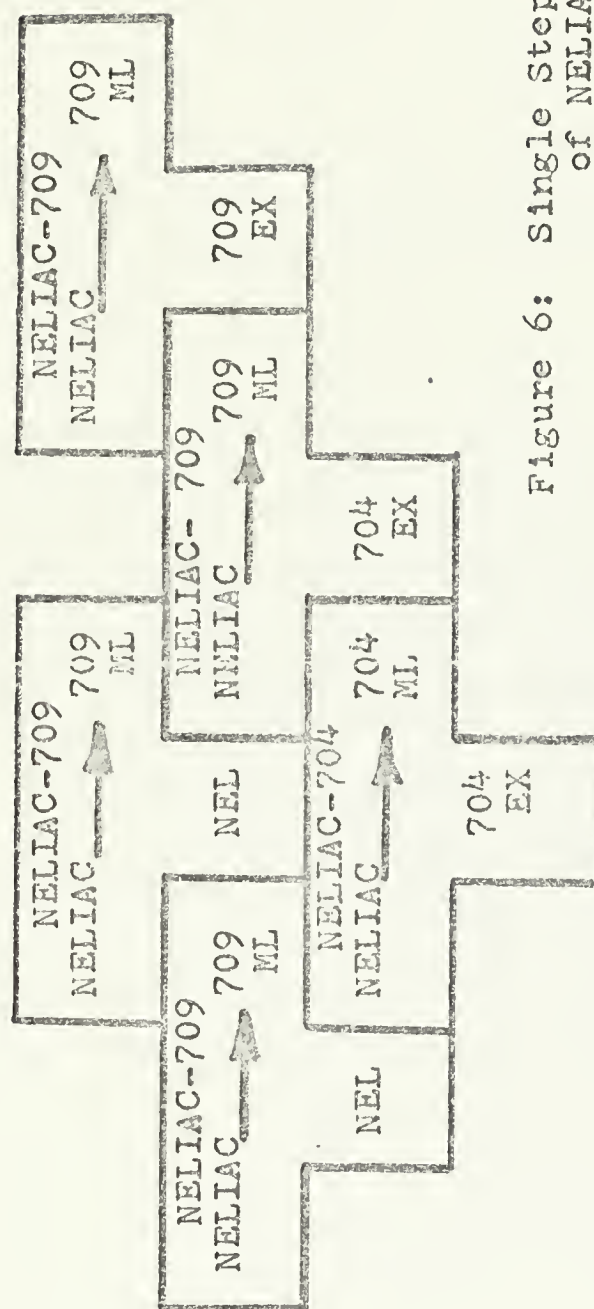
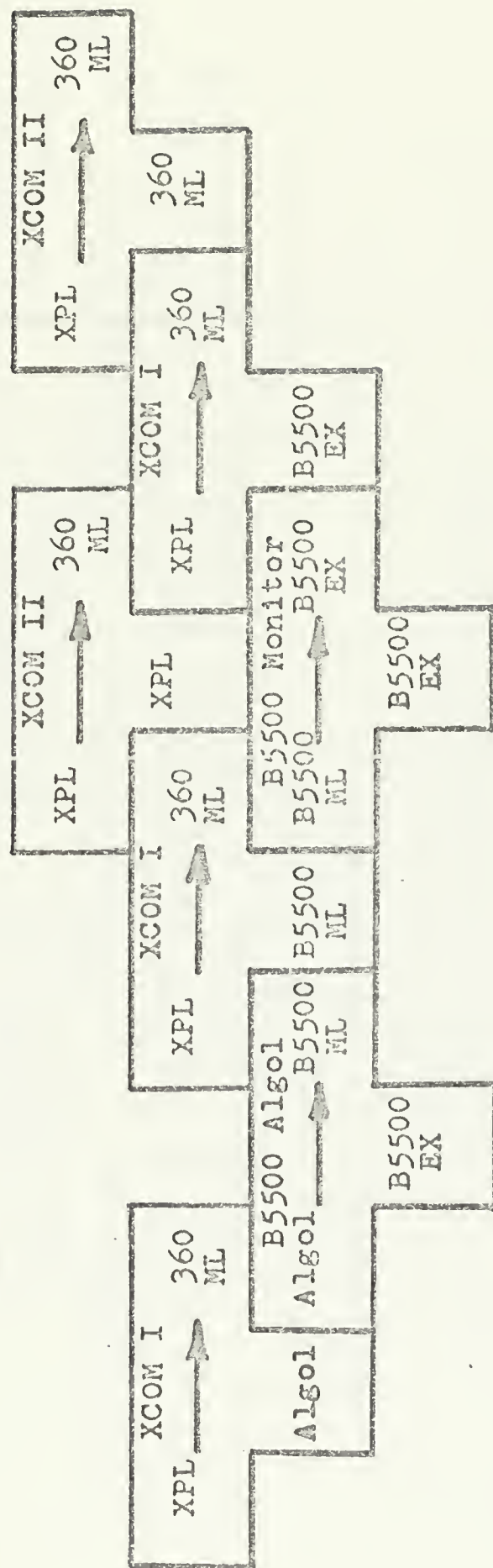


Figure 5: Development of NELIAC
1604 Compiler



implemented on the B5500 computer for their first compiler. This compiler, which could execute the B5500, was used to compile the second version of XCOM which was written in XPL. The product of this second translation was a machine language version of XCOM for the IBM 360 computer, which would compile future additions of the original XCOM compiler. Using a system of abbreviations analogous to those used to describe the Neliac development, Figure 7 illustrates the development of the XPL system for the IBM 360 computer.¹

¹Figure 7 was taken from the inside front cover of Reference 7.



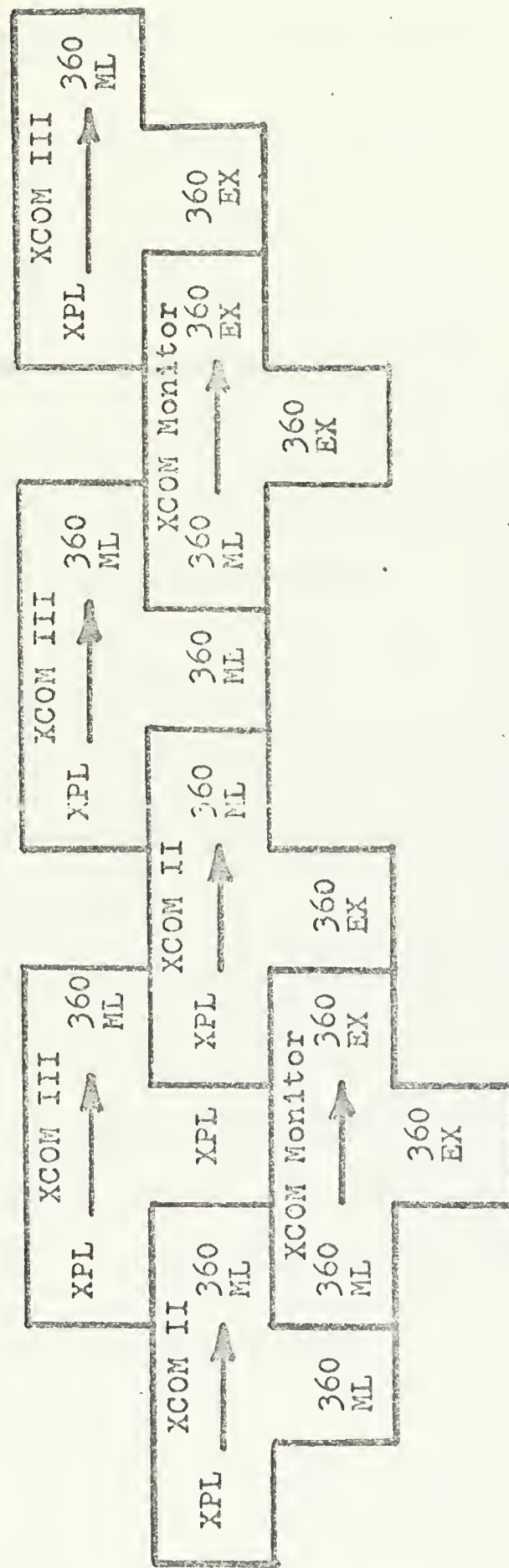


Figure 7 (continued)

VI. GOALS FOR THE FIRST COMPILER

The examples of compiler implementation for the Neliac and XPL-360 systems previously discussed were presented mainly to introduce the several ways in which bootstrapping could be used to assist the compiler writer in developing his system. The bootstrapping used by these compilers illustrate but one of the many tools and methods available which would be fundamental in determining the goals and basic developmental approach taken by the compiler writer.

A. RESOLVING CONFLICTING GOALS

Of the many general constraints encountered in the design of a compiler system, virtually all could be directly or indirectly related to several conflicting goals.

1. The time allowed for compiler development.
2. The efficiency of the resulting code.
3. The degree of compiler efficiency in execution.

To complicate matters further, the measure of "efficiency" used above is again related to the optimal use of several conflicting resources, among the most important being:

1. execution speed, and
2. the main memory used by the compiler.

Several more goals are added to those above by the fact that this compiler must bridge the differences between two machines. In

particular, there is need for sufficient generality to prevent rejection of the compiler on the new system. This rejection could come about in several ways, such as:

1. The use of functions and algorithms which are dependent upon the original machine (e.g., the XPL BYTE function, or "bit strings"), or
2. The dependence upon algorithms which exceed the limitations of the new machine (e.r. main memory requirements).

B. GOALS FOR COMPILER GENERALITY

This need for generality overshadows the goal of optimal efficiency on the first bootstrapping iteration. If the two computers can be successfully bridged with a workable version of the language being bootstrapped (in this case XPL), evolutionary bootstrapping steps on the new computer would then be oriented towards achieving this second goal of compiler efficiency. Stemming from this need for generality the goals for the first compiler eventually took the following form.

1. To use assembler as the target language, since it is flexible enough to efficiently implement any XPL construct and of sufficient generality to provide protection against the first type of rejection, (i.e. reliance upon machine dependent operations).
2. To develop algorithms for the construction of emitted code of sufficient independence and simplicity to allow

their alteration, if it is later found that their use would cause either type of rejection.

3. To minimize the use of compile-time or execution-time optimization as complex algorithms would be required for their implementation. Obviously, some optimization was required by the second restriction, i. e., that of exceeding the limitations of the new machine. The implementation of the XPL system on the CDC 6500 did require the use of several simple optimization algorithms designed specifically to reduce the amount of emitted code and core required for the compiled program. Such concessions to obtain a feasible implementation are further discussed in Chapter VII.

C. THREE PHASES OF DEVELOPMENT

Many subgoals were determined in order to implement the general goals specified for the first compiler. The subgoals tended to divide the development of the XPL-6500 compiler into three main evolutionary bootstrapping steps. The subgoals for each of the three steps which eventually produced the first XPL compiler for the CDC 6500 were:

XCOM-1: develop a translator of sufficient complexity to
 parse any XPL program and emit syntatically
 correct COMPASS assembler statements for all

XPL constructs which required direct code emission to be efficiently implemented.

XCOM-2: build a compiler using XCOM-1 and a set of library procedures sufficient to implement all fixed numeric functions allowable in the XPL language (e. g. BYTE) or implicitly needed to implement XPL (e. g. "bit strings").

XCOM-3: construct a machine independent compiler, written in XPL for the IBM 360/67 and consisting of the previously developed XCOM 2 and a library of sub-routines which would implement all string operations and functions (including input and output of sequential and random files) allowable in the XPL source language.

D. GOALS FOR XCOM 1

Several factors determined the definition of XCOM-1, the first step in the bootstrapping operation. The definition required XCOM-1 to parse an entire XPL program, but not to emit all of the code necessary for its execution. The reasons for these restrictions were simple, and illustrate the power of the evolutionary bootstrapping technique. As XCOM-1 was built upon SKELETON, it inherited the parsing algorithms used by the XPL system. Both this algorithm and the code emission routines as developed for XCOM-1 required the use of many XPL functions which would have been very difficult to

completely implement at this level. Through the use of evolutionary bootstrapping these difficult functions were easily implemented at higher, or later bootstrapping steps. This method was illustrated previously when the history of the Neliac system for the M460 computer was discussed. The objective in that case for the initial step was to obtain a nucleus which was as small as possible or required the least amount of machine level coding, to still be able to compile a workable subset of the total Neliac language. It is this subset through which the next bootstrapping steps could expand the translative abilities of the initial nucleus compiler. The subset of the 48 character XPL language determined sufficient for XCOM-1 required that the following types of statements be implemented by XCOM-1:

1. all storage allocation for numeric and string variables,
2. all fixed numeric and Boolean operations,
3. All branching and conditional statements,
4. procedure definitions and calls, and
5. special calls used internally by the future bootstrapping iterations.

Now that the goals for each step within the bootstrapping process for the XPL-6500 compiler have been discussed it should be clear that they would not yield the optimum compiler. Many of the goals and restrictions determined important for this compiler, such as generality and flexibility, would not retain the same weight once the gap between the two computers had been breached. The re-evaluation of these goals

and the effects of new restrictions would determine a different set of priorities, such as optimal use of time and storage, and would guide the compiler writer in the development of the next compiler. A similar need exists for a two or three step bootstrapping process in order to implement an efficient version of XPL-6500, as it did during the bootstrapping of the XPL system from the Burroughs B5500 computer to the IBM 360. Given this background and these specific goals for the first compiler, its implementation can now be discussed.

VII. IMPLEMENTATION OF THE FIRST COMPILER

The previous sections provided goals for each of the three intermediate compilers needed to effectively bootstrap the XPL system from the IBM 360/67 to the CDC 6500. For each step in the bootstrapping operation many methods and combinations of methods were available to complete each step. Four methods, monitor calls, library procedures, and compiler generated procedures of two types were used to the extent that they warrant further discussion.

A. AVAILABLE METHODS

Two variables appeared to play an important role in determining which of the four methods were used to implement a given task, operation, or function. First the size of the task, measured in both the time needed for execution and the tasks complexity were considered. Second, the frequency that this task was called upon to perform its given function together with the task size would determine which of the four methods would implement it most effectively.

1. Monitor Calls

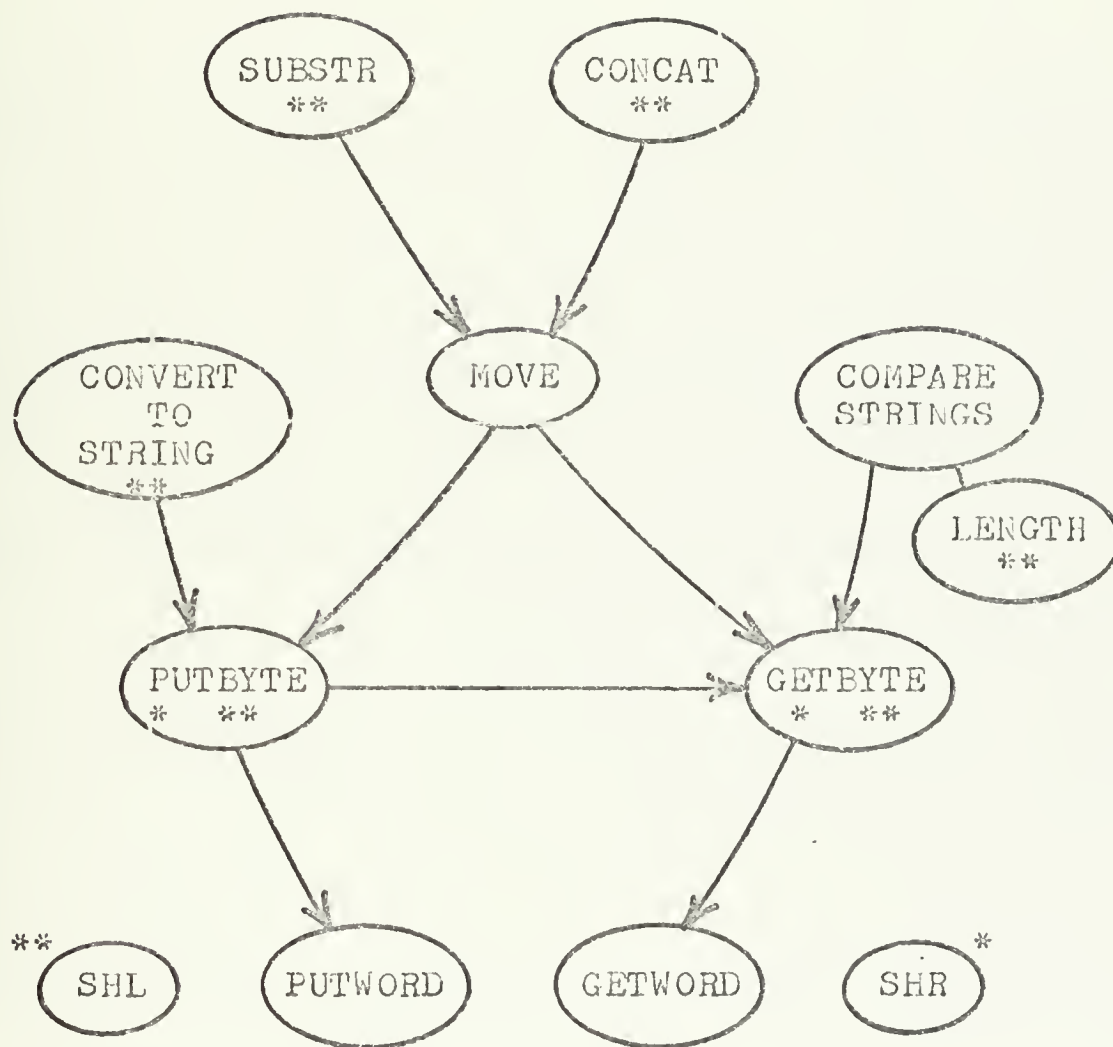
Monitor calls were used to process tasks which were too complex to be emitted from within the compiler and were used too frequently to allow for the inefficiencies of library procedures. These calls were not without cost, as program execution speed would be slowed by the building and passing of parameters and results to and

from the monitor. Interaction with the operating system for input, output, file maintenance, and error recovery procedures for the executing XPL program were found to be most efficiently implemented through monitor calls. This is not to imply that only operating system interaction (the "normal" monitor function) could be directed by the monitor, since any compiler operation could take advantage of the optimized coding within the monitor to maximize their execution speed. The actual monitor developed for use in the XPL 6500 system is discussed in Section C of Chapter VII.

2. Library Procedures

Library procedures which were originally written in the XPL 48 character set language and compiled by XCOM 1 prior to compilation of the user's program, can be used for tasks which are complex and yet infrequently called. Thus, the advantages of writing in a higher level language would not be offset by the inefficiencies of compiler emitted code. Aside from the obvious flexibility provided through the use of library procedures, there is a second advantage to their use with the XPL language. It was found that the initial compiler, XCOM 1 could be greatly simplified through the use of library procedures. This was due to the fact that the same algorithms which processed user defined procedures could be used to call library procedures which would implement almost all of the string functions and the operations needed by the complete XPL language. This is another example of where the power of the bootstrapping technique significantly reduced

Figure 8: Structure of XPL-6500
Library



and simplified the problems to be faced at the machine code or assembly code emission level. Figure 8 illustrates the hierarchy and structure of the libraries developed for the XCOM-2 and 3 bootstrapping steps. These library procedures are listed in Appendix C.

3. Compiler Generated Procedures

Compiler generated procedures were used in the XPL-6500 implementation for simple tasks used with a frequency which would make the necessary monitor linking inefficient. Similarly, these procedures were used for tasks which required special code emission in order to be implemented. In the case of the XPL-6500 implementation it was determined that only four of the library procedures (SHL, SHR, PUTWORD, and GETWORD) were required to be emitted from within XCOM-1. The entire library structure could then be implemented through XCOM-1 and these four built-in procedures.

The compiler generated procedures were separated into two classes. They were considered as either tasks emitted as separate and complete procedures, or as tasks which are integrated within the general code emission algorithm. The first type was chosen to implement the four procedures previously mentioned. Although this approach provided the simplicity and generality desired it was also extremely inefficient, and should be one of the first areas altered in future bootstrapping steps on the CDC 6500.

Although it is not the purpose of this paper to discuss in detail the implementation of the XPL language on the CDC 6500, some detail is

provided to illustrate how the concepts presented in previous sections were employed. This discussion would also highlight the implementational differences between the XPL-6500 and its parent system on the IBM 360 as well as provide an understanding of what the next bootstrapping iteration should include.

B. IMPLEMENTATION OF XCOM-1:

To achieve the desired goals of simplicity and generality for the first compiler, the code generation algorithms developed for XCOM-1 differed radically from those used by the parent XPL 360/67 system. This was to be expected considering the different purposes between the optimized XPL-360 compiler and a compiler designed to bridge two separate machines. The basic rule used to govern the emission of code from the XCOM-1 was to emit code at the earliest time allowable by the syntactic structure. This approach would obviously not allow for the type of optimization achieved in the compiler developed for the IBM 360, where code was emitted at the last possible step. The approach did, however, permit the development of a simple, straightforward compiler which could be altered with a minimum of effort and confusion. Three of the major sections of XCOM-1 are now considered.

1. Code Production and Optimization

The XPL system minimized the difficulties of code emission by designing the parsing algorithm in SKELETON so that it would call upon SYNTHESIZE to emit code for each syntactic production found in

the input text. This capability was inherited by XCOM-1 through the use of SKELETON as a nucleus. Figure 9 illustrates the structure of procedures which developed around SYNTHESIZE to emit COMPASS assembler statements.

The code production algorithms in XCOM-1 were altered in two areas to produce a more efficient and optimized execution code. This optimization was found to be a matter of necessity. It was determined during the development of XCOM-1 that without this additional coding for minimal code optimization XCOM-1 could not compile itself since the self-compilation by an unoptimized XCOM-1 would produce more code than could reasonably be allowed.

For these reasons the code generation algorithms were examined for areas where minimal change would product maximum results. An examination of the BNF representation for the XPL-6500 language (Appendix A) showed that virtually all statements, while being parsed, will use the branches in the syntatic tree leading from $\langle \text{identifier} \rangle$ or $\langle \text{constant} \rangle$ to $\langle \text{expression} \rangle$. It is not surprising then that these heavily traveled syntatic statements were the first areas of the XCOM-1 code generation algorithm which required improvement.

These improvements were in two areas. First a crude form of constant propagation was devised which essentially maintained a table of emitted constants. This algorithm insured that only one location for each unique numeric constant was built, and that all future references to a particular value were to that location. The procedure EMIT

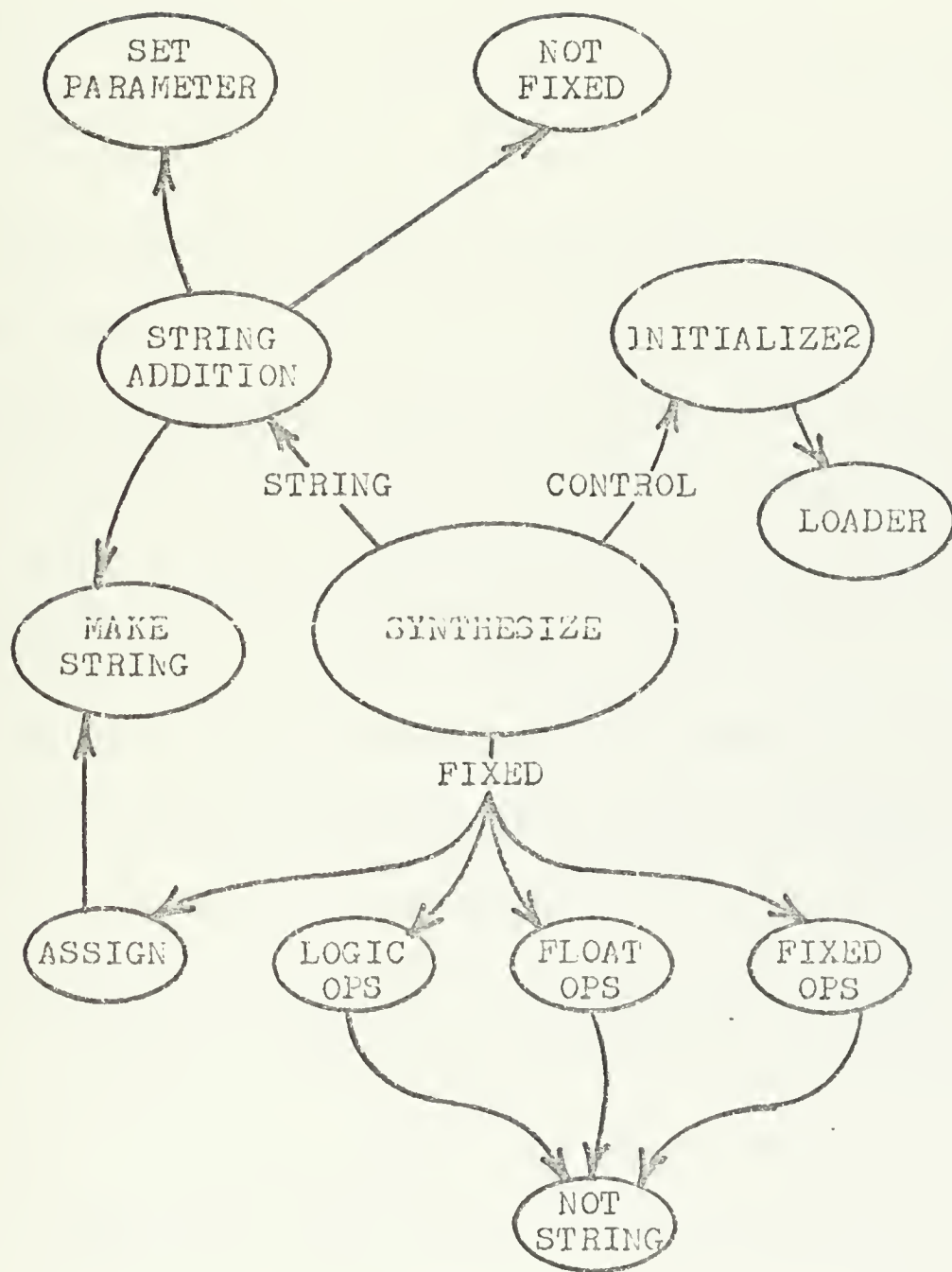


Figure 9: XCOM-1 Code Production Procedures

CONSTANT was designed for this purpose. Aside from this type of constant propagation, constants were also emitted within a central processor command whenever possible. This eliminated the load operation which would otherwise be required.

The effect of the constant propagation algorithm on the volume of unnecessary code was negligible and a more fundamental change was needed. The original algorithm required that all values passing through the syntactic equations leading to $\langle \text{primary} \rangle$ be protected by first creating a new storage location and then storing the value in it. This algorithm was simple and effectively protected all variable and constant locations from being inadvertently destroyed. Through this device future productions did not have to consider variable protection in their algorithms.

Unfortunately, an examination of the emitted code showed that the majority of values passing through this production did not require this protection. Needless storage locations as well as load and store operations were being created. To solve this problem the algorithms for all productions which could possibly destroy the contents of a variable, or constant location were changed. These productions would now determine if the value being altered was the ORIGINAL, or SCRATCH value. If an ORIGINAL value were found, procedure PROTECT would be called to create a new location. Although simple optimization algorithms were already being used (e.g. the reuse of storage locations in arithmetic parsing steps) the effect that this single

global change had on the "quality" of the emitted code was dramatic.

From this single case it can be seen that the leverage obtained by waiting as long as possible before emitting code is enormous and well worth the effort.

2. Code Emission

To emit a COMPASS assembler statement XCOM 1 would produce what was essentially a machine language statement and pass it to a group of procedures which would then construct, format, and emit a COMPASS assembler statement. The production of an assembler statement from a machine language statement, clearly a step backwards, was designed to permit the compiler to bypass the assembly step completely at some future bootstrapping step. Again, the assembly step was needed for this compiler to act as an intermediary so that machine differences (especially word size) could be resolved. A future alteration to XCOM 1 would then be to include all the functions of the assembly step, such as resolution of all symbolic addresses (i. e. , backstuffing addresses), and production of the object module within the compiler proper. The basic structure of the code emitting routines used to produce the COMPASS assembler statements is illustrated in Figure 10.

3. The Symbol Table

To implement the first bootstrapping step only a primitive symbol table was needed. For each symbolic reference used in the XPL program text, a symbol table entry consisting of the following parameters was made:

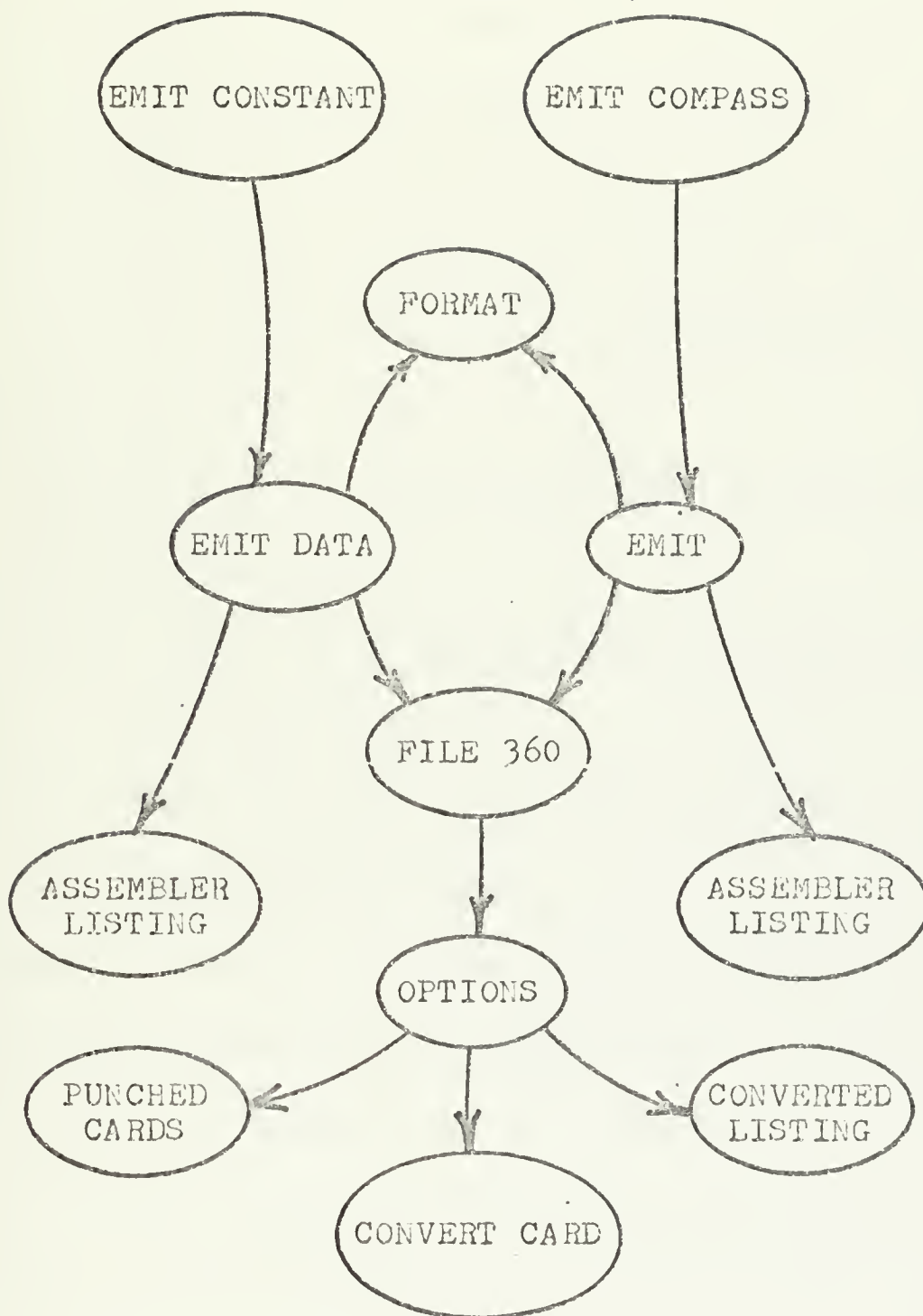


Figure 10: XCOM-1 Code Emission Procedures

1. STYID, which contained the symbolic name of this variable,
2. STYLOC, which contained the label name (or location) of the first word allocated to this symbol,
3. SYTTYPE, contained an integer from 0 to 9 representing the type of symbol located at SYTLOC. Table II describes the symbol types used to implement the XCOM-1 symbol table algorithms,
4. PRAMCNT, indicated the number of parameters declared for this symbol,
5. REFER, which counted the total references made to this symbol table, and
6. DECLARED_ON_LINE, which contained a card reference for diagnostic purposes.

Figure 11 illustrates the basic structures needed to provide normal symbol table functions for the XCOM-1 compiler. The XPL language, since it allows for the declaration of local variables and nested procedures, produces additional requirements for the symbol table procedures. In order to protect these locally declared variables and procedures, it was necessary to add several constructs (PROC_MARK, and LAST_PRAM) to the symbol table algorithm illustrated in Figure 11. The effects that these extra restrictions had upon the over-all symbol table algorithm are detailed in Figure 12. An amply annotated listing of XCOM-1 is provided in Appendix E. To assist in any further work

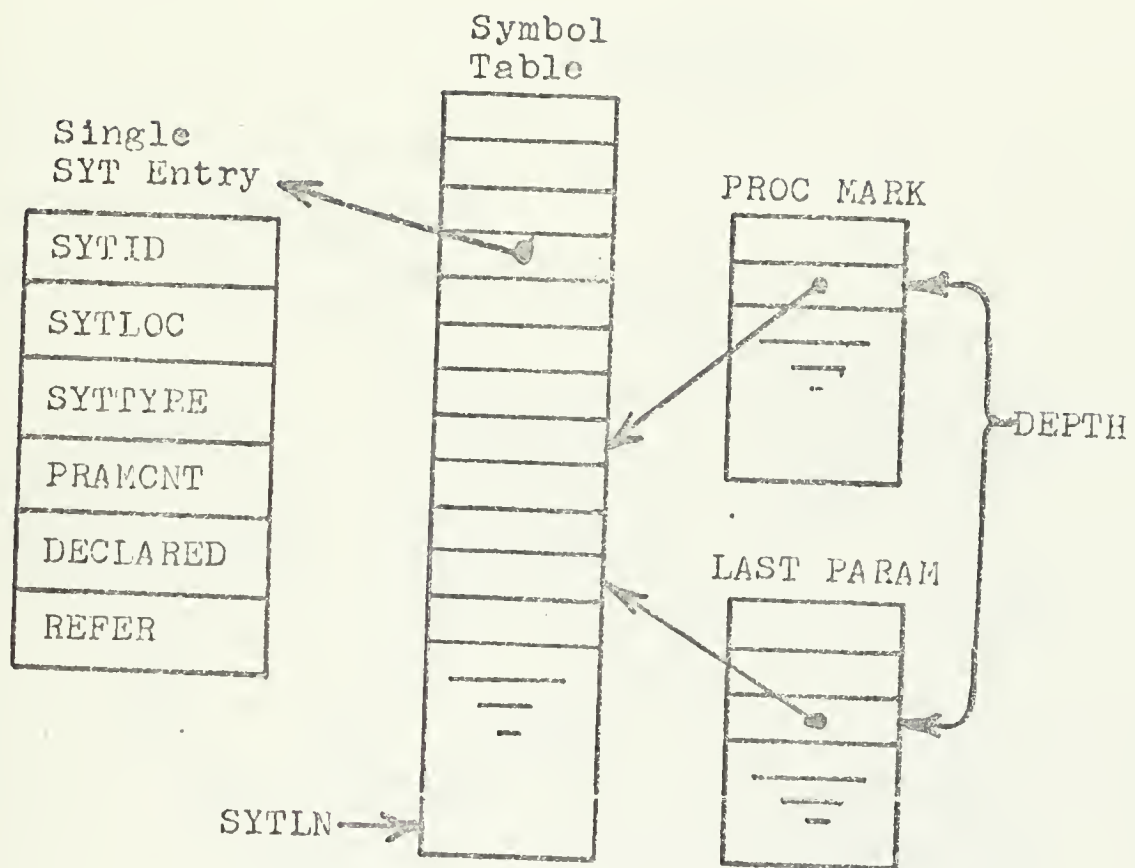


Figure 11: XCOM-1 Symbol Table Structures

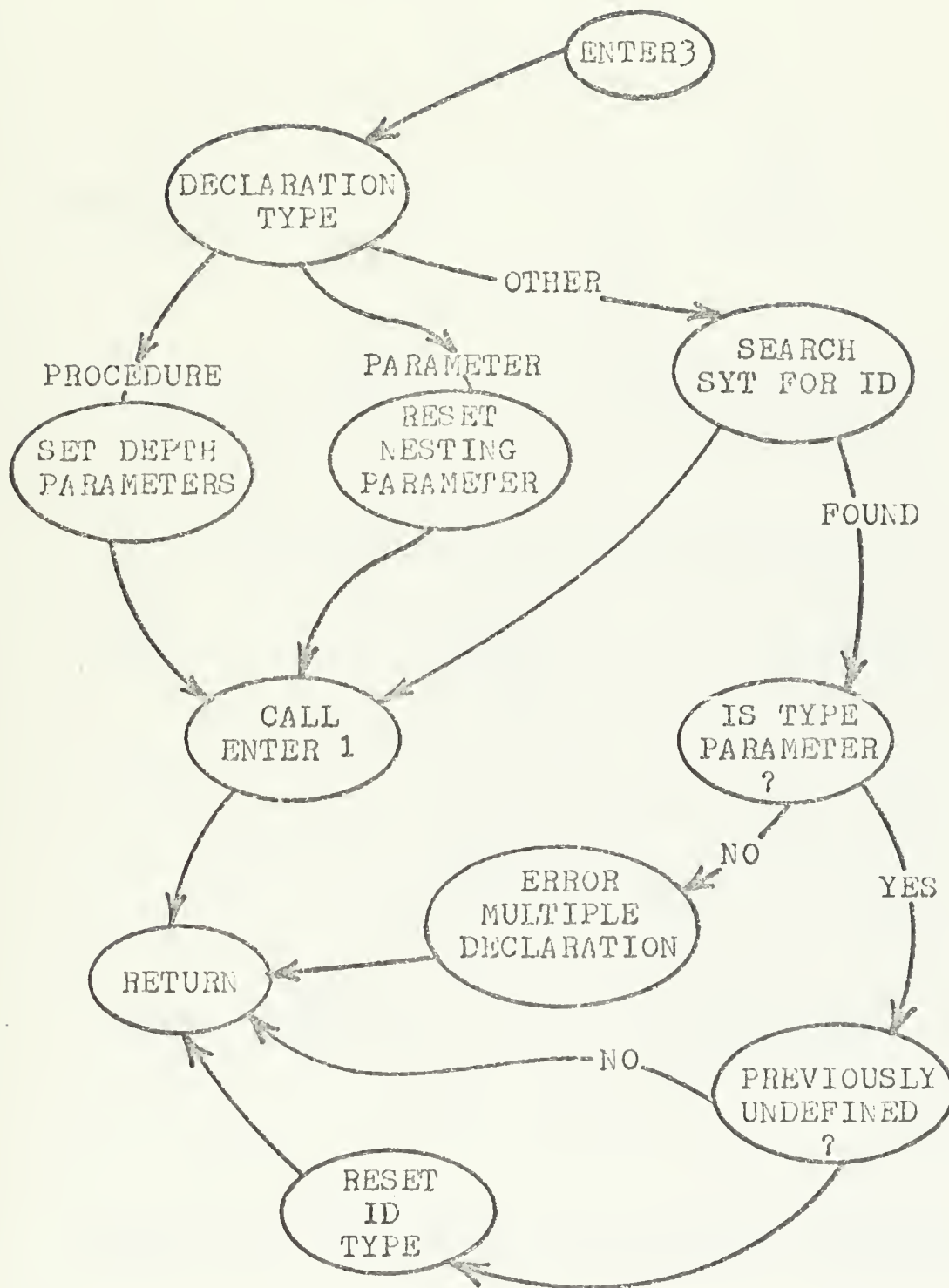


Figure 12: Symbol Table Algorithm for Local Protection

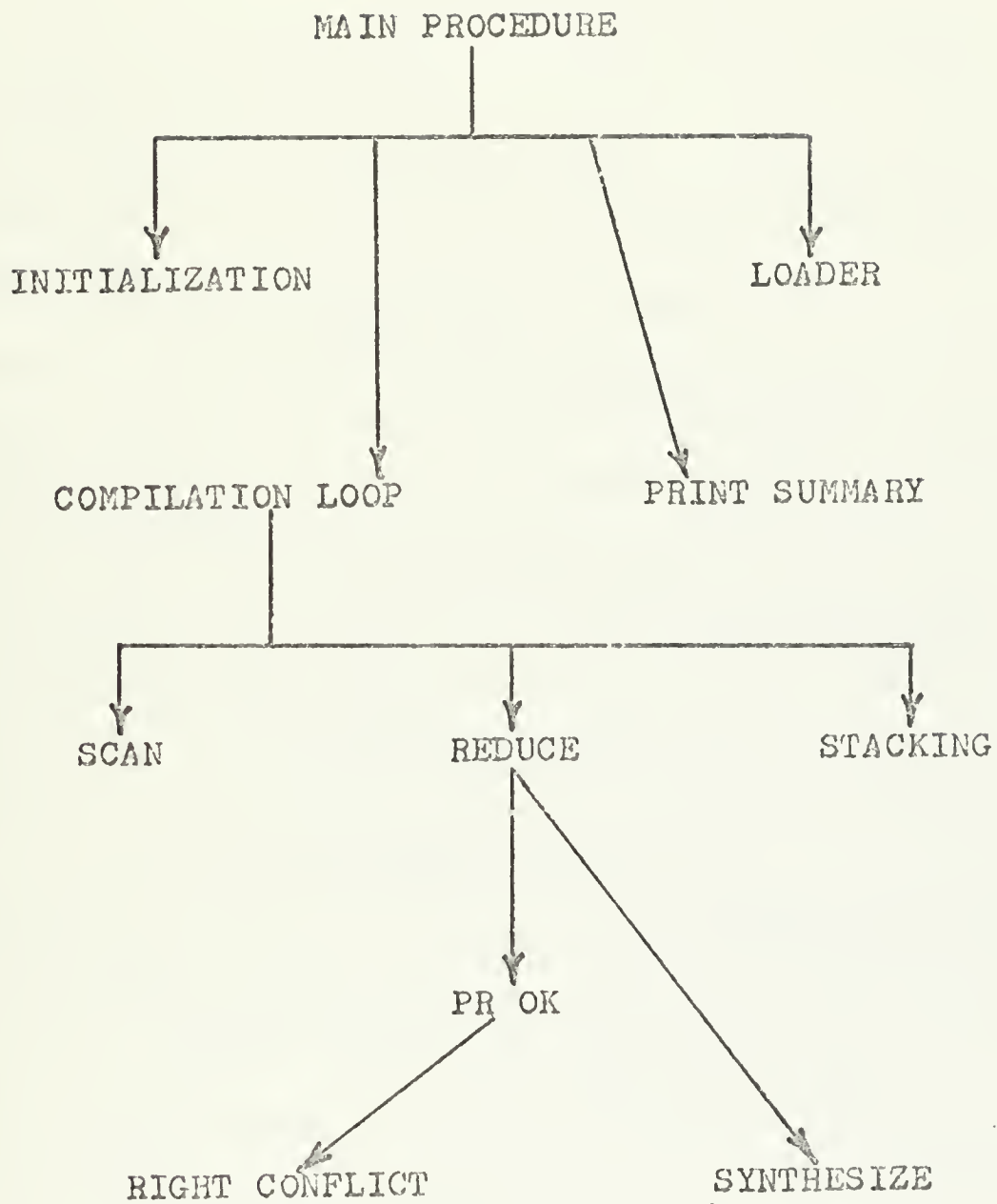


Figure 13: Procedures Provided by SKELETON

on the XCOM-1 compiler, a listing of its major procedures and their functions is provided in Table III.

Many routines were not included within Table III as these procedures originated with SKELETON, the XPL proto-type compiler, and are thoroughly described elsewhere [Ref. 7]. Figure 13 illustrates the relations among the major procedures present in SKELETON down to the point where they interface with SYNTHESIZE.¹

The XPL compiler developed for the IBM 360 worked through a submonitor for the sake of efficient system interaction. A submonitor was also employed in the implementation of the XPL 6500 system. It is discussed here to introduce some of the problems faced by a compiler writer when interacting with an operating system.

C. A MONITOR FOR THE FIRST COMPILER

The monitor's primary function, as previously mentioned, was to simplify the compiler's interaction with the operating system. In the case of the CDC 6500 this operating system was SCOPE 3. A brief discussion of the SCOPE 3 operating system is now presented to provide some insight into the problems encountered while developing the monitor for the XPL-6500 system.

SCOPE 3 is a file oriented operating system which is resident within one of the ten peripheral processors of the CDC 6500. Communication from the executing program to this operating system can be achieved through the use of any other peripheral processor (PP)

¹ Figure 13 taken from Reference 7.

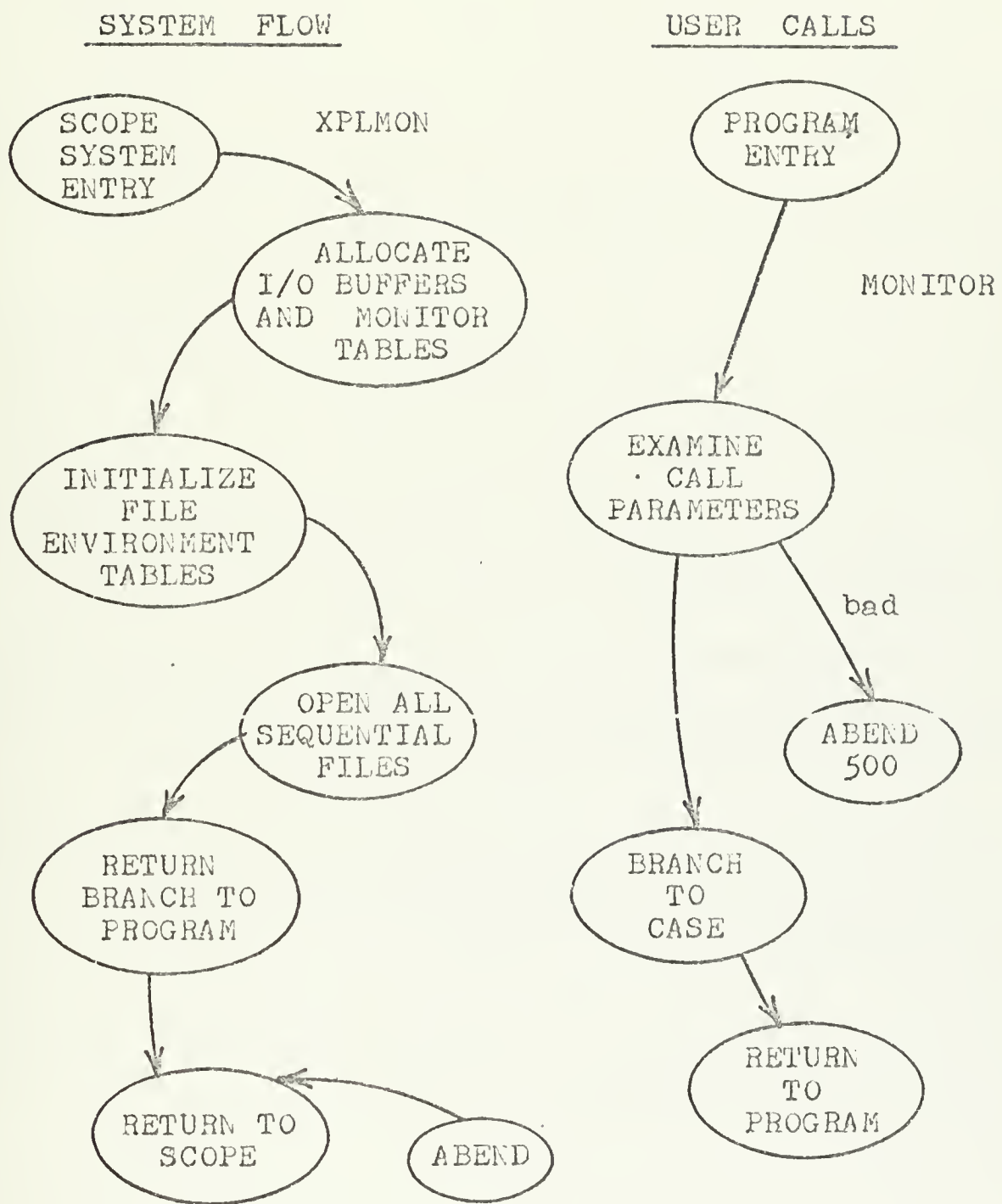


Figure 14: XPL-6500 MONITOR
Program Flow

acting as an intermediary. The monitor for XPL, which is unconcerned as to which peripheral processor is supervising its execution, creates calls to SCOPE 3 in two steps. In the specific case of input, output, and file interaction, the monitor must first construct or update a "File Environment Table" (the FET) with information relative to that call. At a minimum, this updating usually consists of resetting the "Working Storage Address" (WSA) by the "First Word Address" (FWA), the actual core location of the variable(s) involved. The FET is then used as a reference in a two word call placed in a specific location within the monitor which is examined by the PP. All system calls issued by the XPL monitor also employed the "Recall" option which essentially would force the program out of execution (via the CDC Exchange Jump process) until the call had been completed. This feature was used as it significantly improved the CPU utilization.

Once the program was restarted after the system call, the next main Monitor function was to examine the FET for indications of error, and if found, to take action upon them. As the Monitor is presently implemented, this action consists of causing the program to terminate (as in the XPL 360 submonitor). When termination occurred, a descriptive "Completion Code" was passed to SCOPE 3 to be issued and to indicate where in the Monitor the error was found. Error recovery procedures were not implemented for this first Monitor since the XPL 6500 system had not developed to the point where the actions necessary for useful error recovery had been determined. In summary, for each

Monitor-SCOPE 3 interaction involving I/O there were basically three functions to be performed by the Monitor.

1. Update the File Environment Table
2. Call the SCOPE 3 operating system
3. Examine the FET for indications of error.

Since there were two systems using the Monitor as an interface, SCOPE 3 and the executing program, there were two logical program flows through the Monitor, as indicated in Figure 14. Upon first being called by SCOPE 3, the basic assumption was that all sequential input and output files would be used, so they were then created and opened (random files were created dynamically). The Monitor would then call "PROGRAM" into execution (the XPL 6500 Compiler always labeled its final code file as "PROGRAM"), and wait for either the program to terminate, or for a call to "MONITOR." MONITOR was designed to handle eight calls, all involving file interaction. Table IV summarizes the MONITOR calls and their actions.

The monitor, written in COMPASS assembler, was constructed as a large case statement to provide flexibility for alteration and development. Many convenience calls, such as date and time, were not implemented in this version of the monitor since they were not essential for the first compiler's operation. The development of a monitor which includes these calls should be considered during the next bootstrapping step of XCOM. The internal logic of the monitor is provided in Appendix B. This appendix provides a base of complete documentation for future

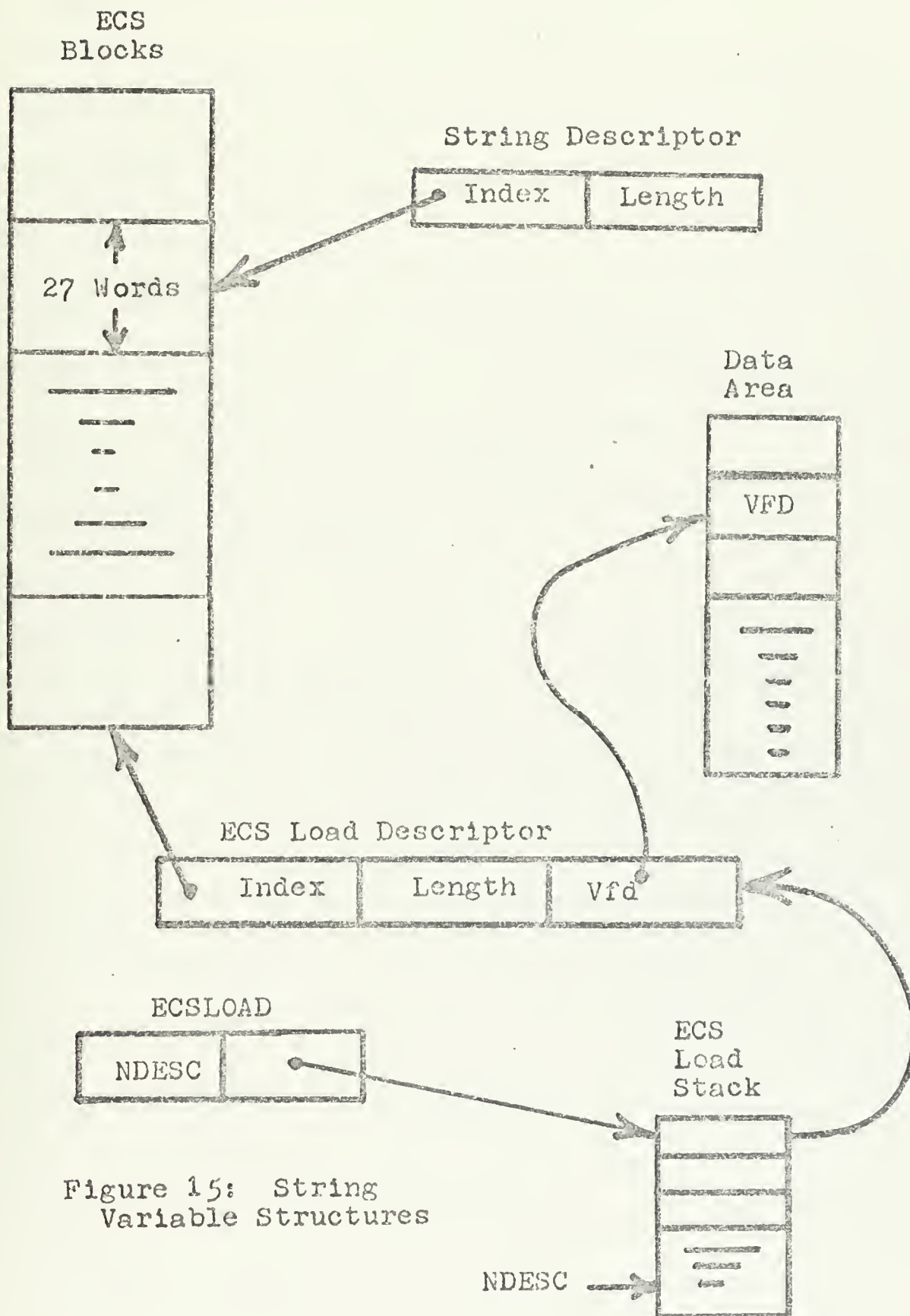


Figure 15: String Variable Structures

additions and alterations. The COMPASS assembler listing of the first XPL system submonitor is contained in Appendix D.

D. IMPLEMENTATION OF STRING VARIABLES

The implementation of constant and variable strings, normally a significant problem, was easily achieved through the application of bootstrapping techniques. These techniques together with the goal of simplicity for the first compiler produced an algorithm significantly different from the original XPL 360/67 string implementation. A discussion of the XPL 360/67 string algorithm is presented to illustrate the differences between a dynamic string algorithm and the static XPL 6500 algorithm.

1. XPL 360/67 Implementation

The XPL 360/67 compiler system employs a dynamic string storage algorithm. This algorithm, although complex and expensive in execution time, provides a method for efficient storage utilization. All character strings are addressed, or referenced indirectly through "string descriptors," which are words containing both the character string length and its first word location in the string area. These string descriptors are stored in what is essentially an array (actually a "page" designated for descriptors) and are referenced by string variables through their location within this array. To perform any string operation during execution the descriptor represented by a string variable value, would first be loaded. The character string,

or substring defined by this descriptor would then be operated upon. Thus the only storage statically fixed during execution would be the actual string descriptor array.

This string descriptor array would also give, at any given moment during execution, an accurate picture of all active locations in the string area. Thus a program (the XPL-360/67 COMPACTIFY) could examine this descriptor array and compact the active storage without altering the pointers and other links between variable names and the actual character strings.

The problems which result from the use of this type of algorithm are twofold. First the compiler would have to build all the necessary structures and links between symbolic names and descriptor locations. In addition the compiler would have to store the extra information needed to execute the dynamic string algorithm. This would be a compile time problem. During program execution any character string operation would have to work through this table of descriptors. This would also force all submonitor-program interaction for sequential input and output to operate through this table.

It was determined after a significant effort towards implementing a dynamic string algorithm for the XPL-6500 system, that the benefits gained through such an algorithm were not worth their attainment. A much simpler and efficient algorithm for static character string storage allocation was developed for the XPL-6500 system. This static algorithm

was further simplified through the use of two previously mentioned methods; bootstrapping and library procedures.

2. Use of Bootstrapping and Library Procedures

The implementation of a static string allocation algorithm for the XPL-6500 system was achieved and greatly simplified through the use of library procedures and bootstrapping techniques. Although simpler in design than the dynamic algorithm discussed, the static allocation algorithm developed for the first step, if implemented within XCOM 1, would have required a higher degree of complexity and rigidity than could be tolerated. Library procedures were used to perform all string operations which did not demand direct code emission for their implementation. These procedures provided an algorithm which achieved the flexibility, simplicity, and clarity that was determined necessary for the first compiler.

The technique of evolutionary bootstrapping was also used to further simplify the implementation of the static string allocation algorithm. This technique allowed XCOM-1 to develop as a separate compiler for all fixed numeric statements without being altered significantly as the string algorithm developed. As discussed in Chapter VI, XCOM 1 was used to compile itself as a first step. Library procedures were then used to produce a compiler capable of all XPL string operations and functions.

3. String Variable Structures and Operations

Although the algorithms used to implement the string variables in the XPL 360/67 and XPL-6500 systems differed radically, their basic structures did not. The static string allocation algorithm developed for XPL-6500 was designed to act as an interface between XCOM-1 and a one million word Extended Core Storage (ECS) capability available to the CDC 6500 computer. Several factors required that this ECS capability be used, the primary being the size of the static string array.

The XPL language allows variable length strings to have a maximum size of 256 characters. It was determined that to compile just the string variables of XCOM-1, assigning fixed blocks of 256 (10 characters per 60 bit word) for each string variable, more central memory storage would be required than was available on the CDC 6500 (131K words).

Two avenues were then available to solve this problem. First, all the programs written for the XPL 360/67 implementation could be rewritten for a fixed string length algorithm, or the ECS capability could be used. The fixed string length approach was determined the best, but would require rewriting all the XPL-360 programs used in the bootstrapping effort. In order to avoid this, the ECS approach was taken.

These blocks of ECS were referenced indirectly through descriptors similar to those used by XPL 360/67. Descriptors were used to include string length information which would be needed by

most string operations. Aside from the string length, these descriptors contained the first word location in ECS of the desired fixed string block. To perform any string operation a monitor call was issued to load in a specific string block. Upon completion of the operation, the resulting string could be stored in ECS through another monitor call. This arrangement provided a simple and efficient system for storing and retrieving strings. The efficiency was gained from the time saved moving character strings, as the ECS device could transfer blocks of core storage (larger than 25 words) at a faster rate than central memory. This procedure would implement half of the string variable problem. The implementation of constant character strings, declared at compile time, required separate development.

To initialize the ECS fixed string blocks with the constant character strings found during compilation, a second descriptor type was needed. As the constant strings were encountered in the program text, code was emitted to store them in the variable data area. At this time a descriptor (ECSLOAD) was produced which contained the location in the data area of the string (a label name), the length of the string, and the ECS block that the string should be stored in before execution. At the end of compilation, all these ECSLOAD descriptors were placed in the variable data area, and the load file was written containing a monitor call to initialize the ECS blocks with the character strings represented by the ECSLOAD descriptors. Figure 15

summarizes the structures developed to implement a static character string algorithm for XPL 6500.

VIII. CONCLUSION

Although the present need for optimal compiler operation limits the production application of bootstrapping and compiler generator languages, the future need for special purpose languages in specific problem areas should demand their use.

The histories of the NELIAC and XPL-360/67 compiler systems, as well as the simplification of the compiler project for the CDC 6500, illustrated the power and effectiveness of bootstrapping techniques. A close examination of these compiler generator languages should reveal that it is not the powerful instructions, or complex algorithms contained within these systems that alone produce their strength. Rather, the efficacy of these new compiler generator languages is created from the ability to employ bootstrapping techniques.

TABLE I

IBM 360 AND CDC 6500 CHARACTER SET COMPARISON

<u>CHARACTER</u>	<u>PUNCH CODE</u>		<u>BYTE CODE</u>	
	<u>360</u>	<u>6500</u>	<u>360</u>	<u>6500</u>
A	12-1		C1	01
B	12-2		C2	02
C	12-3		C3	03
D	12-4		C4	04
E	12-5		C5	05
F	12-6		C6	06
G	12-7		C7	07
H	12-8		C8	10
I	12-9		C9	11
J	11-1		D1	12
K	11-2		D2	13
L	11-3		D3	14
M	11-4		D4	15
N	11-5		D5	16
O	11-6		D6	17
P	11-7		D7	20
Q	11-8		D8	21
R	11-9		D9	22
S	0-2		E2	23
T	0-3		E3	24
U	0-4		E4	25
V	0-5		E5	26
W	0-6		E6	27
X	0-7		E7	30
Y	0-8		E8	31
Z	0-9		E9	32
0	0		F0	33
1	1		F1	34
2	2		F2	35
3	3		F3	36
4	4		F4	37
5	5		F5	40
6	6		F6	41
7	7		F7	42
8	8		F8	43
9	9		F9	44

TABLE I

IBM 360 AND CDC 6500 CHARACTER SET COMPARISON

CHARACTER	PUNCH CODE		BYTE CODE	
	360	6500	360	6500
-		11	60	46
*		11-8-4	5C	47
/		0-1	61	50
\$		11-8-3	5B	53
blank		space	40	55
,		0-8-3	6B	56
.		12-8-3	4B	57
:		8-2	7A	63
+		12-8-6, 12	8E	45
(12-8-5, 0-8-4	4D	51
)		11-8-5, 12-8-4	5D	52
=		8-6 , 8-3	7E	54
>		0-8-6 , 11-8-7	6E	73
<		12-8-4, 12-0	4C	72
¬		11-8-7, 12-8-6	5F	76
≡		*****, 0-8-6	**	60
[*****, 8-7	**	61
]		*****, 0-8-2	**	62
≠		*****, 8-4	**	64
→		*****, 0-8-5	**	65
√		*****, 11-0	**	66
^		*****, 0-8-7	**	67
↑		*****, 11-8-5	**	70
↓		***, 11-8-6	**	71
≤		*****, 8-5	**	74
≥		*****, 12-8-5	**	75
&		12 , *****	50	**
%		0-8-4 , *****	6C	**
?		0-8-7 , *****	6F	**
#		8-3 , *****	7B	**

TABLE I

IBM 360 AND CDC 6500 CHARACTER SET COMPARISON

<u>CHARACTER</u>	<u>PUNCH CODE</u>		<u>BYTE CODE</u>	
	<u>360</u>	<u>6500</u>	<u>360</u>	<u>6500</u>
@	8-4	*****	7C	**
'	8-5	*****	7D	**
	12-8-7,	*****	4F	**

TABLE II. DESCRIPTION OF VARIABLE TYPES

<u>XCOM 1 Code</u>	<u>Symbol</u>	<u>Meaning</u>
0	UNKNOWN	the variable type has not been defined
1	LABELTYPE	a branch location
2	PARAMETER	a procedure parameter of unspecified type. This label is used to prevent storage allocation where parameter is declared
3	FIXEDTYPE	an integer (60 bit) variable or single dimension array
4	CHRTYPE	a string variable or single dimension string array which contains a descriptor
5	PROC	the entry point into a procedure which does not return a value
8	FIXEDPROC	the entry point into a procedure which returns an integer (60 bit) value
9	CHRPROC	the entry point into a procedure which returns a string descriptor

TABLE III. MAJOR XCOM-1 PROCEDURES

A. CODE EMISSION ROUTINES

- | | |
|-------------------|---|
| 1. EMIT COMPASS: | produces the three fields needed for any COMPASS central processor statement from a machine language statement. |
| 2. EMIT CONSTANT: | maintains a table of previously emitted constants and produces new DATA locations for new constants. |
| 3. EMIT DATA: | issues an assembler statement to the DATA area. |
| 4. EMIT: | issues an assembler statement to the CODE area. |
| 5. FORMAT: | builds a single COMPASS statement from three assembler fields. |
| 6. FILE 360: | provides necessary DATA and CODE file maintenance. |
| 7. OPTIONS: | controls the punching and listing of converted BCD COMPASS assembler statements. |
| 8. CONVERT CARD: | provides EBCDIC to BCD conversion |

B. CODE GENERATION ROUTINES

- | | |
|---------------------|---|
| 1. SYNTHESIZE: | produces and controls the production of a machine language-like statement for the code emission routines. |
| 2. STRING ADDITION: | produces code for all implicitly declared string operations and functions. |

- | | | |
|----|----------------|---|
| 3. | MAKE STRING: | builds a call to CONVERT TO STRING which provides numeric to string conversion. |
| 4. | LOGIC OPS: | builds code for AND, OR logical operations. |
| 5. | FLOAT OPS: | builds code for division and multiplication operations through central processor floating point operations. |
| 6. | FIXED OPS: | builds code for addition and subtraction operations using fixed point operations. |
| 7. | ASSIGN: | builds code for variable and variable list assignments. |
| 8. | SET PARAMETER: | builds various calls to library procedures. |

C. SYMBOL TABLE ROUTINES

- | | | |
|----|------------|---|
| 1. | ENTER 3: | controls the declarations of any identifier or string. |
| 2. | CUTBACK: | resets symbol table parameters to remove local variables. |
| 3. | ENTER 1: | builds an initial three part symbol table entry. |
| 4. | SYTSEARCH: | returns the symbol table location of a given identifier. |

D. INITIALIZATION ROUTINES

- | | | |
|----|-----------------|---|
| 1. | INITIALIZATION: | sets all compiler variables necessary to initiate scanning and code emission. |
| 2. | INITIALIZE 2 : | resets compiler variables after library procedures have been compiled. |

3. EMIT BUILT IN CODE: emits code for initial COMPASS assembler interaction and procedures SHL, SHR, GETWORD, and PUTWORD.

TABLE IV. MONITOR CALL SUMMARY

<u>Code</u>	<u>Action</u>	<u>Monitor Call Parameters</u>
0	USER REQUESTED ABEND	(0)
1	GET SEQUENTIAL INPUT	(1, FWA)
2	PUT SEQUENTIAL OUTPUT	(2, FWA, FILE)
3	READ RANDOM FILE	(3, FWA, FILE, RECORD, FILE SIZE, INDEX SIZE)
4	WRITE RANDOM FILE	(4, FWA, FILE, RECORD)
5	READ FROM ECS	(5, FWA, ECS, FWA)
6	WRITE INTO ECS	(6, FWA, ECS, FWA)
7	INITIALIZE ECS	(7, FWA, NUMBER OF DESCRIPTIONS)

BNF G XE PNE JER 6A50I N RESENC TADJ TMA HPN DING NAD EARG OF

[illegible]


```

IDENTIFIER LIST> ::= ( IDENTIFIER LIST> <IDENTIFIER> , */
<IDENTIFICATION> ::= <IDENTIFIER> */ <IDENTIFIER> ) */
<IDENTIFICATION> ::= SPECIFICATION> { */
<BOUND HEAD> ::= <IDENTIFIER> <NUMBER> */
<BOUND HEAD> ::= <IDENTIFIER> <TYPE> */
<TYPE DECLARATION> ::= <IDENTIFIER> <SPECIFICATION> */
<TYPE DECLARATION> ::= <BOUND HEAD> <TYPE> */
<INITIAL> ::= INITIAL TYPE DECLARATION> <INITIAL> <NUMBER> */
<INITIAL HEAD> ::= <INITIAL HEAD> <NUMBER> */
<INITIAL HEAD> ::= <INITIAL DECLARATION> <INITIAL> <STRING> */
<INITIAL HEAD> ::= <INITIAL HEAD> , <STRING> */
<BIT HEAD> ::= BIT ( */
<TYPE> ::= FIXED ACTER */
<TYPE> ::= LABEL */
<TYPE> ::= <NUMBER> ) DECLARATION> */
<DECLARATION STATEMENT> ::= <TYPE ELEMENT> */
<DECLARATION STATEMENT> ::= <IDENTIFIER> LITERALLY <STRING> */
<DECLARATION STATEMENT> ::= <DECLARATION STATEMENT> , <DECLARATION ELEMENT> */
GO TO> ::= GO TO */
GO TO> ::= GO TO > IDENTIFIER */
REPLACE> ::= */
REPLACE> ::= <VARIABLE> , <REPLACE> <EXPRESSION> */
ASSIGNMENT> ::= <LEFT PART> <ASSIGNMENT> */
LABEL DEFINITION> ::= <LEFT PART> <ASSIGNMENT> , . */

```


APPENDIX B. MONITOR FLOW DIAGRAMS

<u>Figure</u>	<u>MONITOR Code</u>	<u>Function</u>
B1	1	Sequential Input
B2	2	Sequential Output
B3	3	Read or Create a Random Access File
B4	4	Write to a Random Access File
B5	5	Read from Extended Core Storage
B5	6	Write to Extended Core Storage
B6	7	Initialize Extended Core String Area

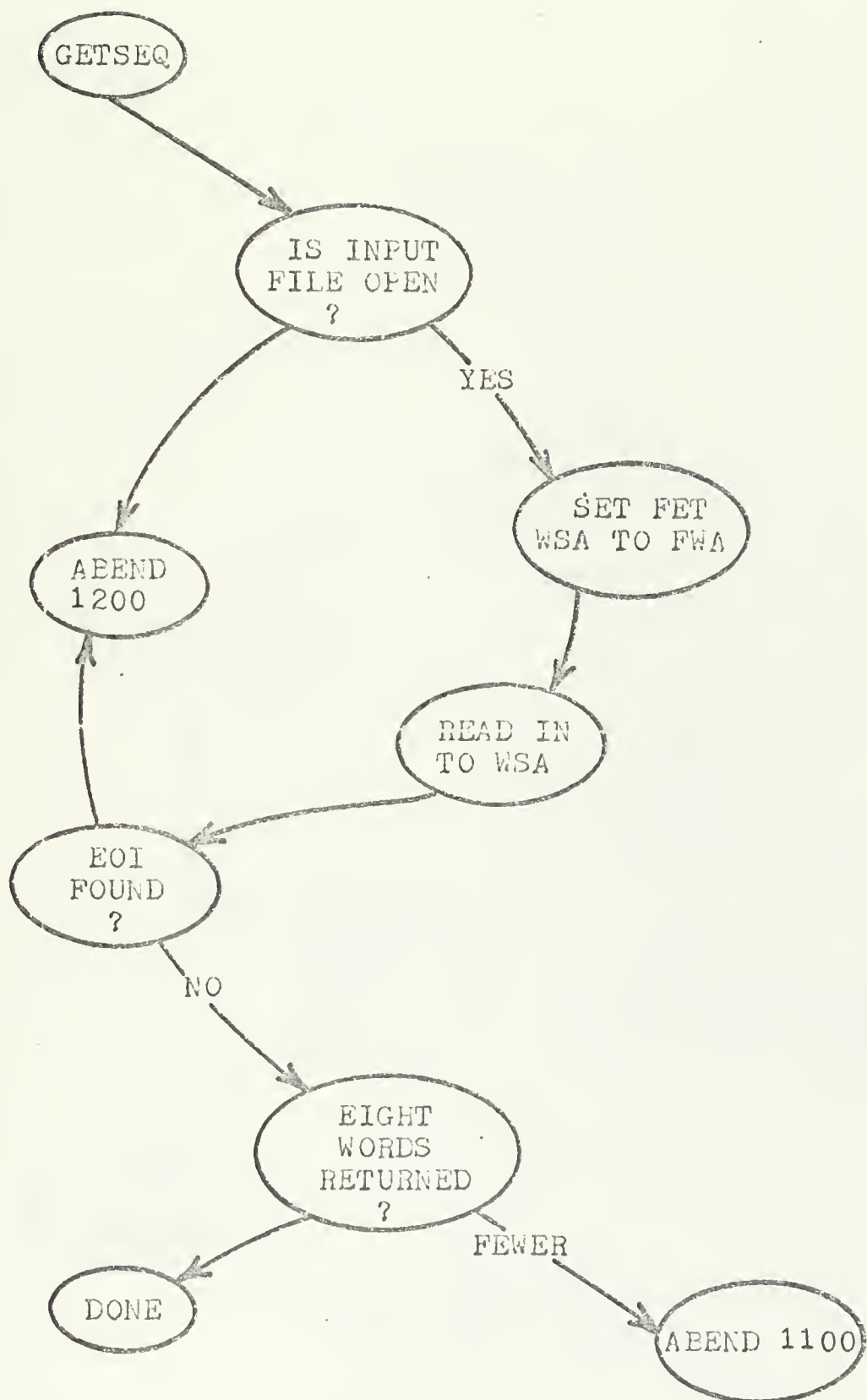


Figure B1: Monitor action for Sequential Input.

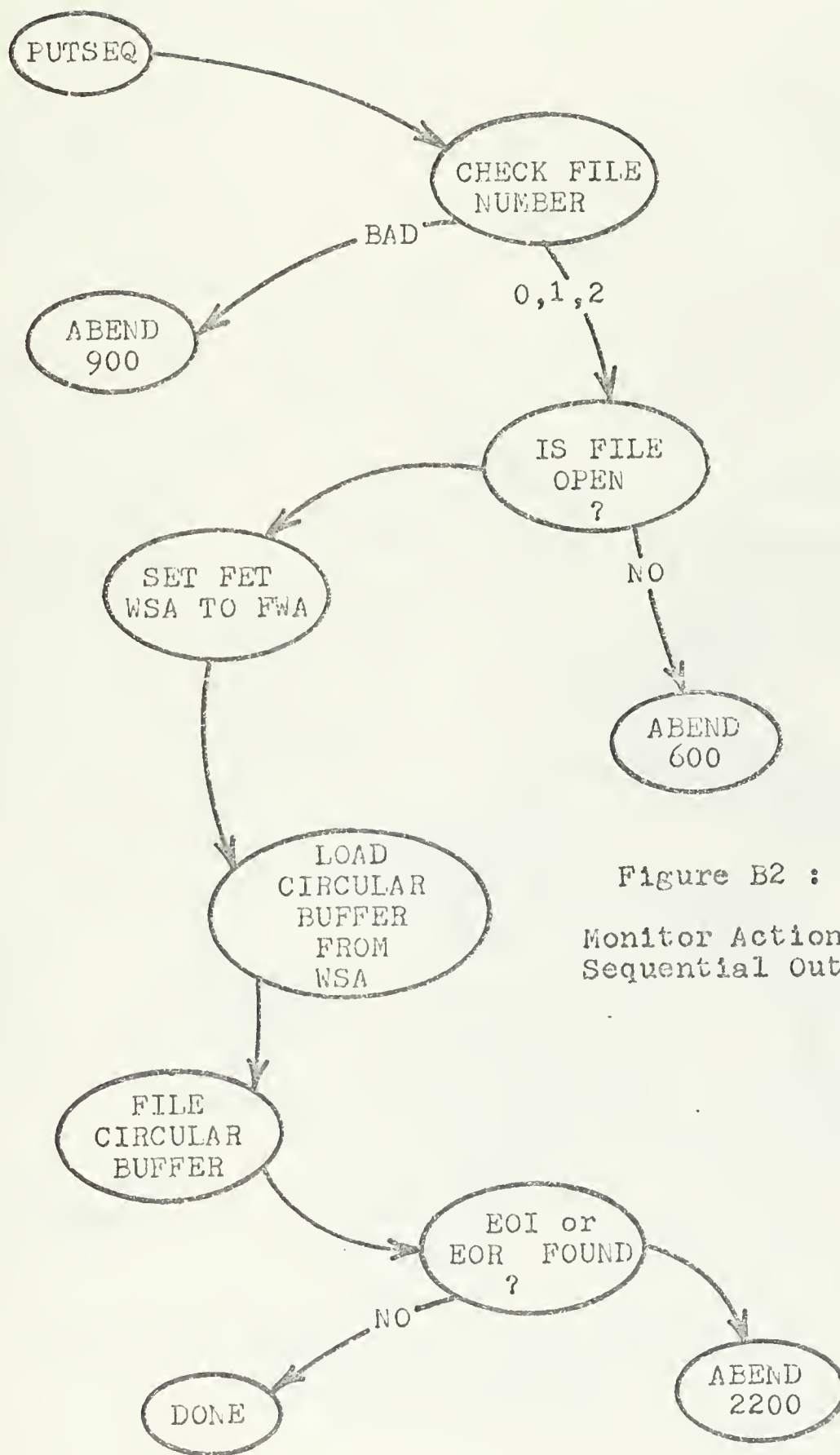


Figure B2 :
Monitor Action for
Sequential Output

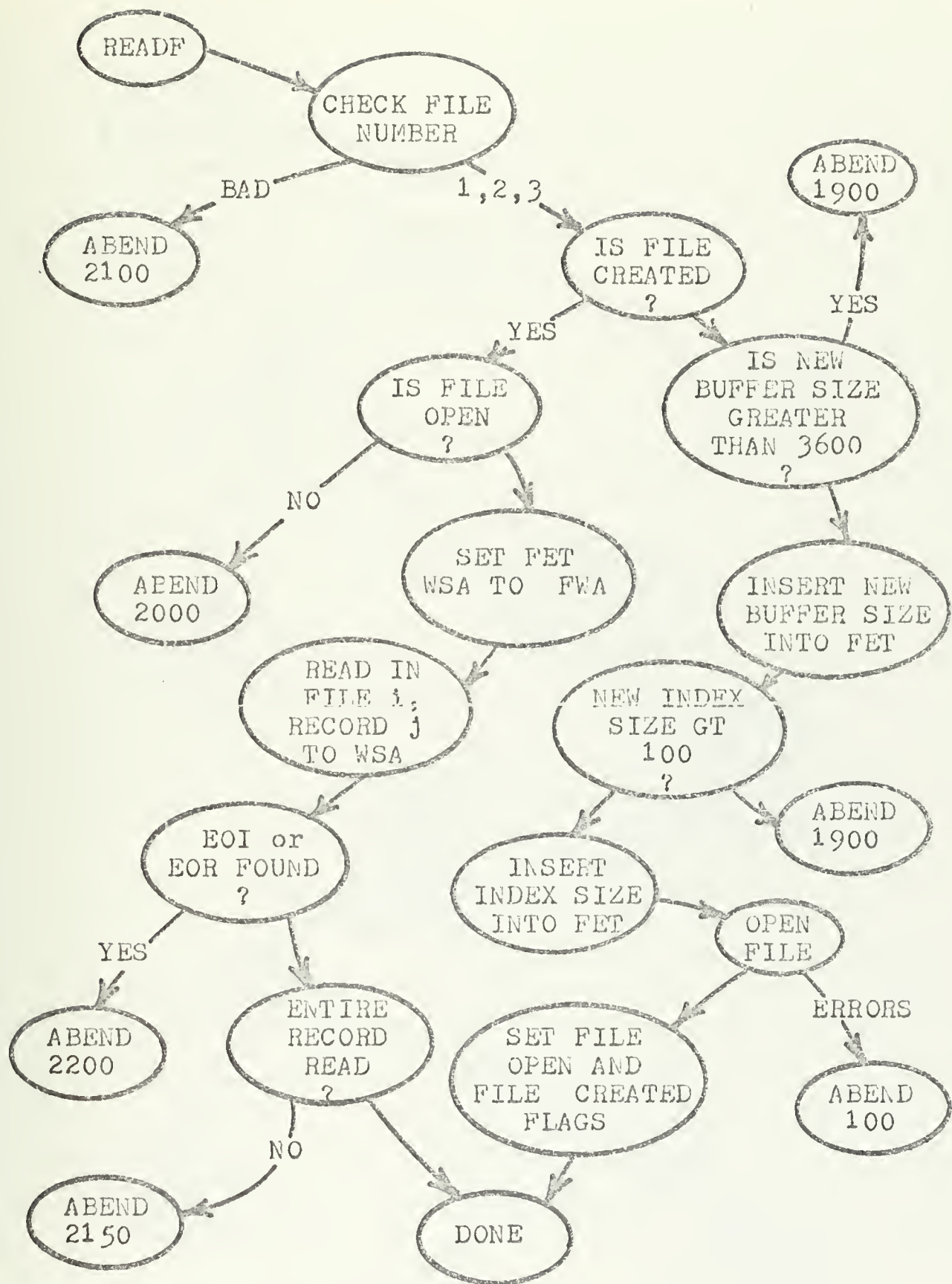


Figure B3: Monitor Action for Random File Reading and Creation.

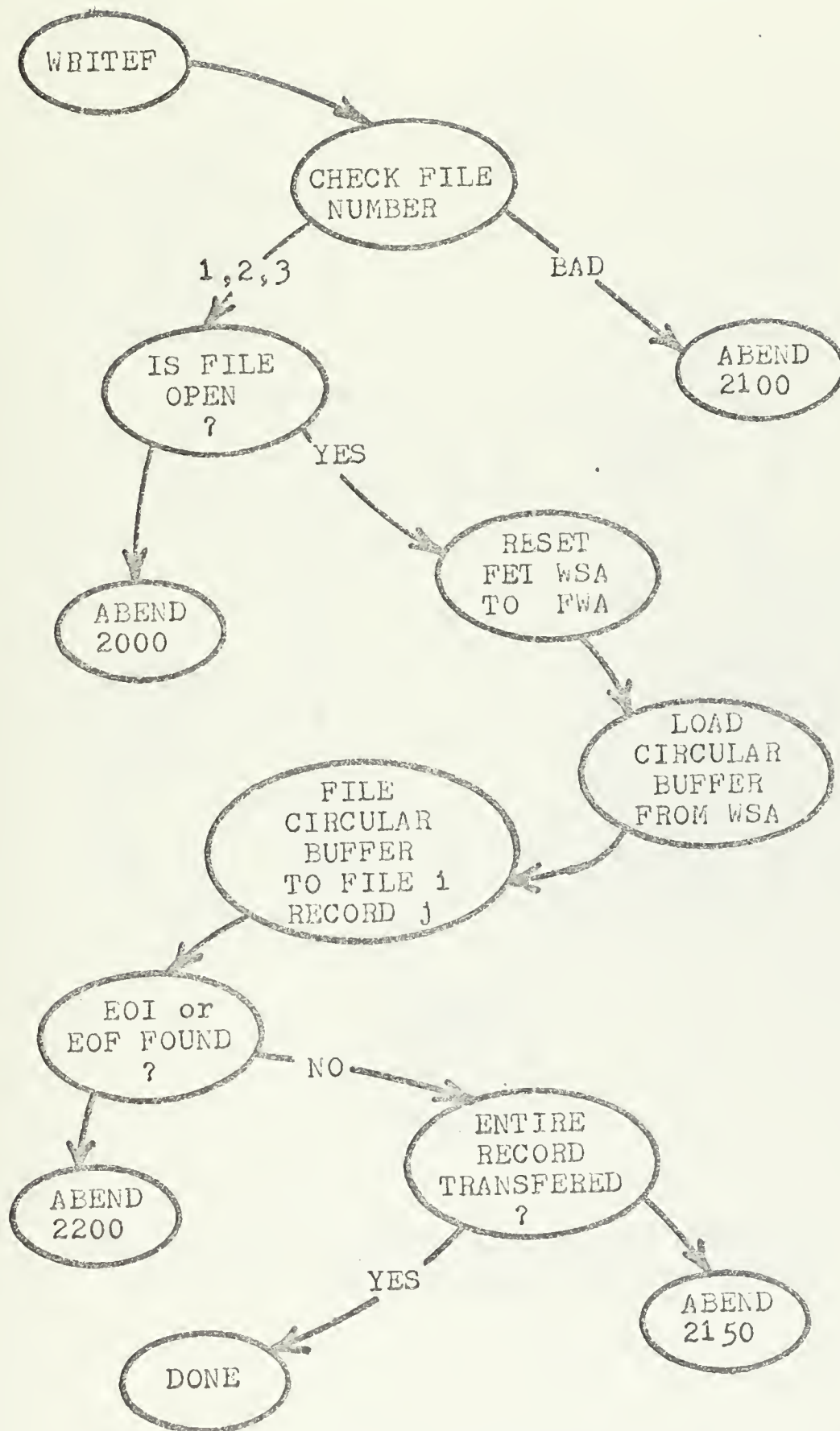


Figure B4: Monitor Action for Random File Write.

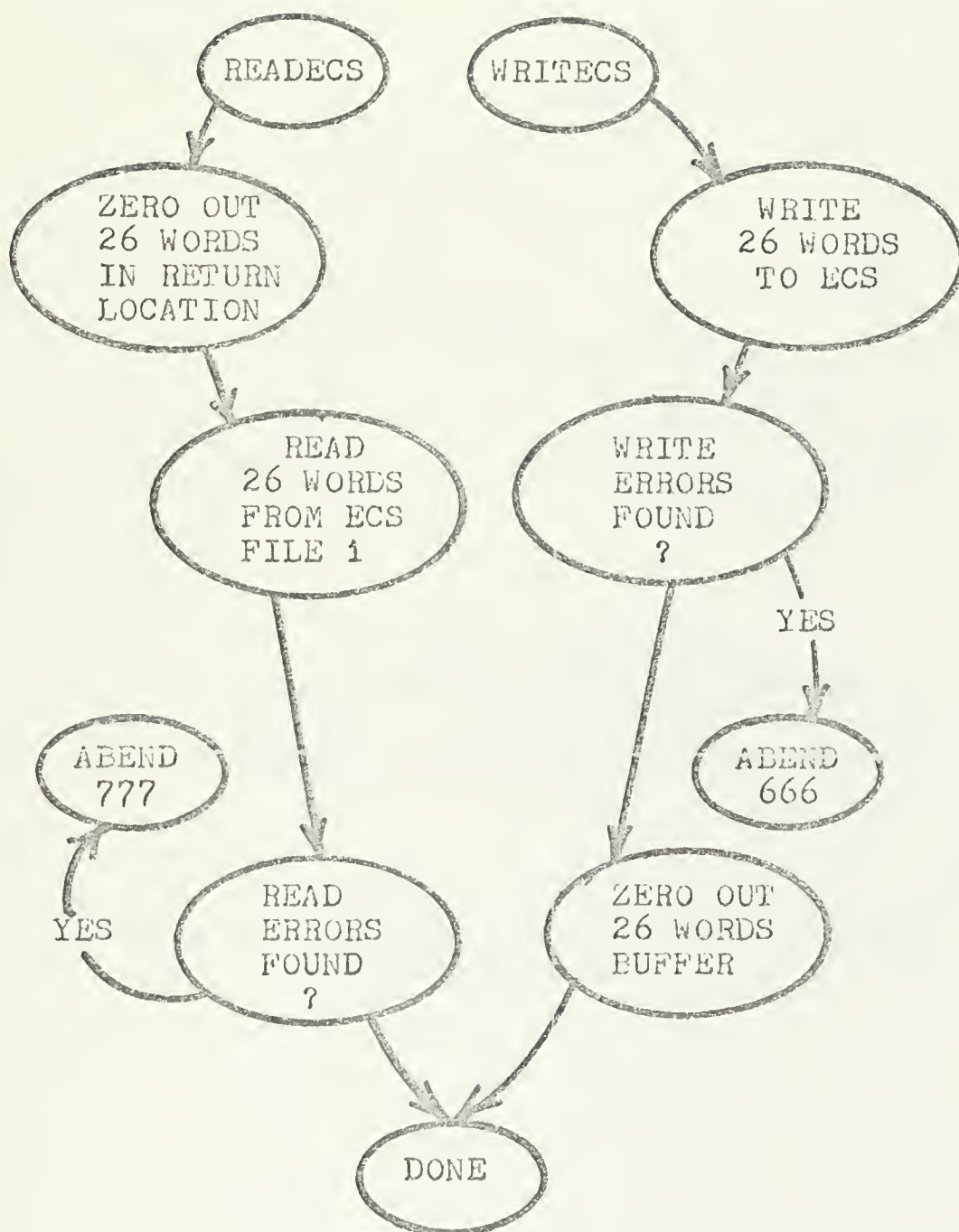


Figure B5: Monitor Action for Extended Core Storage Read or Write.

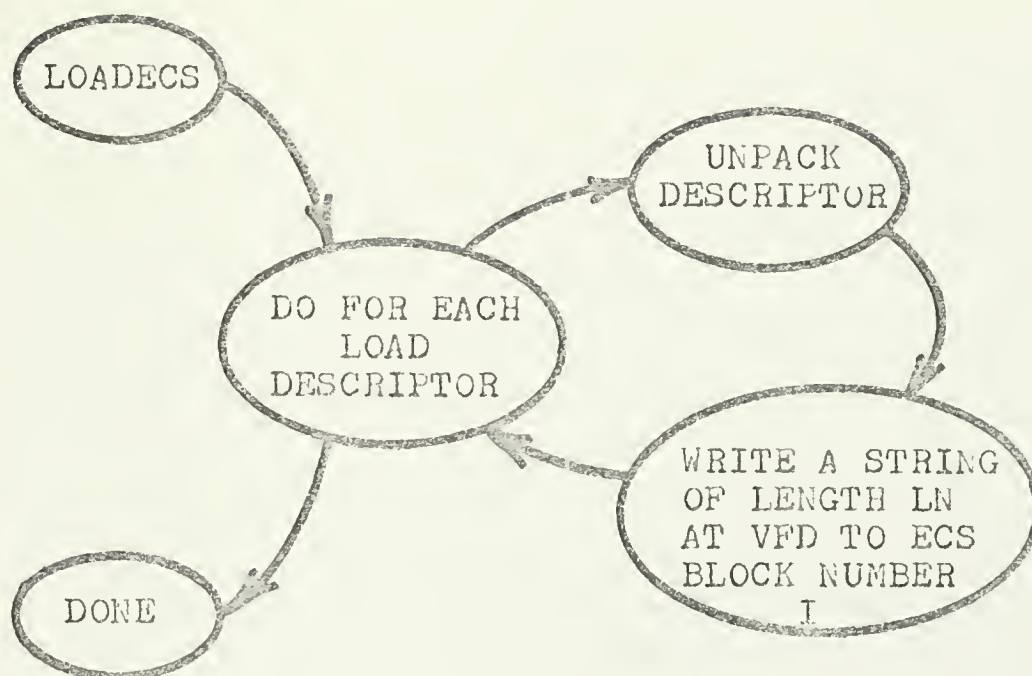


Figure B6: Monitor Action for Initialize ECS String Area.


```

MCNITOR PROCEDURE(CALL NO,P1,P2,P3,P4,P5) FIXED ,
DECLARE(CALL NO,P1,P2,P3,P4,P5) FIXED ,
/* DUMMY ROUTINE TO COLLECT PARAMETERS FOR MONITOR CALLS */
END MONITOR ;

```

```

GETBYTE PROCEDURE(WORD,POS) FIXED ,
DECLARE(WORD,POS,TEMP,MASK) FIXED ,
DO WHILE WORD + 1 ,
WORD = WORD + 1 ,
POS = POS + 1 ,
END
POS = 6 * (9 - POS)
TEMP = GETWORD(WORD)
IF POS NE 0 THEN
DO ,
MASK = $3FF$ , MASK, POS) ,
MASK = SHL(MASK, MASK) ,
TEMP = SHR(TEMP, POS) ,
END
RETURN(TEMP) ,
END GETBYTE

```

```

DECLARE MAP_TO_360 (58) FIXED INITIAL(
/* 0 1 2 3 4 5
** ** ** ** **
** ** ** ** **
** ** ** **
** ** ** **
** ** ** **
*/
0 193, 194, 195, 196, 197, 198, 199, 200,
1 193, 194, 195, 196, 197, 198, 199, 200,
2 193, 194, 195, 196, 197, 198, 199, 200,
3 193, 194, 195, 196, 197, 198, 199, 200,
4 193, 194, 195, 196, 197, 198, 199, 200,
5 193, 194, 195, 196, 197, 198, 199, 200,
6 193, 194, 195, 196, 197, 198, 199, 200,
7 193, 194, 195, 196, 197, 198, 199, 200,
8 193, 194, 195, 196, 197, 198, 199, 200,
9 193, 194, 195, 196, 197, 198, 199, 200,
*/

```

```

BYTE
PROCEDURE(DESC, POS) FIXED ,
DECLARE(DESC, POS, TEMP) FIXED ,
TEMP = GETBYTE(DESC)
IF TEMP LE 57 THEN
RETURN(MAP_TO_360(TEMP)) ,
ELSE
RETURN(240) ,
END BYTE
/* BYTE CODE FOR 360 ZERO */

```



```

LENGTH PROCEDURE(DESC1, FIXED , ,
DECLARE DESC1, DESC2, LN, TEMP)
RETURN(SHR(DESC, 18)) , ,
END LENGTH , ,

COMPARE STRINGS(DESC1, DESC2, LN, TEMP) , ,
PROCEDURE(DESC1, DESC2, LN, TEMP) , ,
DECLARE(DESC1, DESC2, LN, TEMP) , ,
DO TEMP=0 TO 26
SI(TEMP), S2(TEMP)=0, ,
END , ,
LN=LENGTH(DESC1), ,
TEMP=LENGTH(DESC2), ,
RETURN(0) , ,
IF LN MONITOR(5, S1, DESC1 AND $3FFFF$) , ,
CALL MONITOR(5, S2, DESC2 AND $3FFFF$) , ,
TEMP=0 , ,
DO WHILE TEMP LE LN , ,
IF GETBYTE(S1, TEMP) NE GETBYTE(S2, TEMP) THEN
RETURN(1) , ,
END , ,
RETURN(1) , ,
END COMPARE_STRINGS , ,

/* PROCEDURES WHICH RETURN DESCRIPTORS */
/* TURN ON ASSEMBLER LISTING */
/* $E */

PUTBYTE PROCEDURE(DATA, OFFSET, WORD) , ,
DECLARE(DATA, OFFSET, WORD, TEMP) , ,
DO WHILE OFFSET GE 10 , ,
OFFSET = OFFSET - 10 , ,
WORD = WORD + 1 , ,
END , ,
OFFSET = 6 * (9 - OFFSET) , ,
DATA = SHL(DATA AND $3FF$, OFFSET) , ,
TEMP = GETWORD(WORD) AND (-SHL($3FF$, OFFSET)) , ,
CALL PUTWORD(DATA AND TEMP, WORD) , ,
END PUTBYTE , ,
/* $E */
/* TURN OFF UNCONVERTED LISTING */
/* $E */
NOVE PROCEDURE(GETLOC, GETPOS, PUTLOC, PUTPOS, GETLN) , ,

```



```

DECLARE(GETLOC,GETPOS,PUTLOC,PUTPOS,GETLN) FIXED ,.
DO WHILE GETLN GT 0 ,.
IF GETPOS GE 10 THEN
DO ,.
GETLOC = GETLOC+ 1 ,.
GETPOS = 0 ,.
END
IF PUTPOS GE 10 THEN
DO ,.
PUTLOC = PUTLOC + 1 ,.
PUTPOS = 0 ,.
END
CALL PUTBYTE(GETLOC,GETPOS),PUTPOS,PUTLOC) ,.
GETPOS = GETPOS + 1 ,.
GETLN = GETLN - 1 ,.
END
END MOVE ,.

DECLARE NEW_INDEX FIXED ,.          /* LOCATION STUFFED BY XCOM */
/* $H */ /* TURN ON CONVERSION */
/* $J */ /* TURN ON PUNCHING OF CONVERTED CARDS */

CONVERT TO STRING ,.
PROCEDURE(N) ,.
DECLARE(N,NEG,R,Q,I,POS) FIXED ,.
DO I = 0 TO 26 ,.
STRING(I) = 0 ,.
END
IF N EQ 0 THEN
DO ,.
CHAR(I) = 33 ,.
I = I+1 ,.
END ,.
ELSE
DO ,.
LT 0 THEN
IF N NEG = I ,.
N = -N ,.
END ,.
DO ,.
R, Q = N ,.
I = 0
DO WHILE Q NE 0 ,.
Q = Q / 10 ,.
R = R - (Q*10) ,. /* REMAINDER */
IF R GT 4 THEN R=R+5 ,.
I = I+1
CHAR(I) = 33+R ,.
R = Q ,.
END ,.

```



```

IF NEG THEN
DO ,.. CHAR(I) = 44 ,.. I = I+1 ,.. END ,..
END ,..
POS, STRING = 0 ,..
DO WHILE I > 0 ,..
CALL PUTBYTE(CHAR(I),POS,STRING) ,..
I = I-1 ,..
END
CALL MONITOR(6,STRING,NEW_INDEX) ,.. /* WRITE ECS */
RETURN(SHL(I,18)) ,..
END CONVERT_TO_STRING ,..

/* $J */ /* TURN OFF CARD PUNCHING */
/* $H */ /* TURN OFF CONVERSION TO BCD */

SUBSTR PROCEDURE(DESC, START, LN) FIXED ,..
DECLARE(DESC, START, LN, LN1, T) FIXED, S1(27) FIXED, S2(27) FIXED ,..
DO T = 0 TO 26 ,..
S1(T), S2(T) = 0 ,..
END ,..
IF LN EQ 0 THEN LN1 = LENGTH(DESC) ,.. ELSE LN1=LN+START ,..
CALL MONITOR(5, S1, DESC AND $3FFFFFF$) ,.. /* READ FROM ECS */
CALL MOVE(S1, START, S2, 0, LN1) ,.. /* WRITE ECS */
CALL MONITOR(6, S2, NEW_INDEX) ,.. /* WRITE ECS */
RETURN(SHL(LN, 18) + (NEW_INDEX AND $3FFFFFF$)) ,..
END SUBSTR ,..

CONCAT PROCEDURE(DESC1, DESC2) FIXED ,..
DECLARE(DESC1, DESC2, LN1, LN2) FIXED, S1(27) FIXED, S2(27) FIXED ,..
DO LN1 = 0 TO 26 ,..
S1(LN1), S2(LN1) = 0 ,..
END ,..
CALL MONITOR(5, S1, DESC1 AND $3FFFFFF$) ,.. /* READ ECS */
CALL MONITOR(5, S2, DESC2 AND $3FFFFFF$) ,..
CALL MOVE(S2, 0, S1, LN1, LN2) ,..
LN1 = LENGTH(DESC1) ,.. LN2 = LENGTH(DESC2) ,..
CALL MONITOR(6, S1, NEW_INDEX) ,.. /* ECS WRITE THE NEW STRING */
RETURN(SHL(LN1+LN2, 18) + (NEW_INDEX AND $3FFFFFF$)) ,..
END CONCAT ,..

INPUT PROCEDURE(FILE) FIXED ,..
DECLARE(FILE, T) FIXED, S(27) FIXED ,..
DO T=0 TO 26, S(T) = 0 ,.. END ,..

```



```
CALL MONITOR(1,S) ,.. /* GET SEQUENTIAL */
CALL MONITOR(6,S,NEW_INDEX),.. /* WRITE ECS */
RETURN(SHL(80,18) + TNEW_INDEX AND $3FFFF$) ,..
END INPUT ,..

OUTPUT PROCEDURE (FILE ) ,..
DECLARE FILE FIXED ,..
/* DUMMY PROCEDURE TO COLLECT OUTPUT PARAMETERS AND
TRIGGER A MONITOR CALL */
END OUTPUT ,..

PUT_SEQ ..
PROCEDURE(DISC,FILE) ,..
DECLARE(DISC,T,FILE) ,.. FIXED , S(27) FIXED ,..
DO T = 0 TO 26 ,.. S(T) = 0 ,.. END ,.. /* LOAD IN STRING */
CALL MONITOR(5,$,DESC AND $3FFFF$) ,.. /* PUT SEQUENTIAL OUTPUT */
CALL MONITOR(2,S,FILE) ,..
END PUT_SEQ ,..

/* THIS TRIGERS END OF LIBRARY */ /* $Z */

DO_NOTHING ..
PROCEDURE ,..
DECLARE(I,J,K) ,.. FIXED ,..
END DO_NOTHING ,..

/* THIS TRIGERS END OF COMPILING */ EOF
EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF
```


IDENT XPLMON
ENTRY MONITOR, XPLMON
EXT PROGRAM

X P L M O N
S U B - M O N I T O R
F O R X P L - 6 5 0 0
C O M P I L E R S Y S T E M

BY
RONALD C. SMEDER
USN POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA
JUNE 1971

THIS LISITNG OF VERSION 2 OF XPLSM IS
DATED 7 JUNE, 1971

XPLSM WAS DESIGNED AS AN INTERFACE BETWEEN XCOM-ONE
AND THE SCOPE 3 OPERATING SYSTEM FOR THE CONTROL DATA
6500 COMPUTER.

A DISCUSSION OF THE HISTORY AND ALGORITHMS USED
BY BOTH XCOM-ONE AND THIS SUB-MONITOR CAN BE FOUND IN
THE THESIS ENTITLED
AN INVESTIGATION INTO COMPILER GENERATION
BY BOOTSTRAPPING TECHNIQUES

BY
RONALD C. SMEDER
USNPGSCOL
JUNE 1971

THE SUB-MONITOR IS DESIGNED TO HANDEL 8 CALLS

MONITOR CODE	ACTION
0	USER REQUESTED ABEND
1	GET SEQUENTIAL INPUT FROM SPECIFIED FILE
2	PUT SEQUENTIAL OUTPUT TO SPECIFIED FILE
3	READ FROM SPECIFIED RANDOM FILE
4	WRITE TO A SPECIFIED RANDOM FILE
5	READ FROM EXTENDED CORE STORAGE
6	WRITE TO EXTENDED CORE STORAGE
7	INITIALIZE EXTENDED CORE STORAGE FOR STRING OPERATIONS

*** BUFFER SIZE PARAMETERS

```

MAXCODE      EQU      7
RWSAS        EQU      360
SEQBS        EQU      1600
RNDDBS       EQU      365
RNDIS        EQU      100
IWSAS        EQU      8

```

BUFFER STORAGE LOCATIONS

```

RNDWSA      BSS
RNDDB1      BSS
RNDDB2      BSS
RNDDB3      BSS
PBUF        BSS
OUTBUF      BSS
INBUF       BSS
IOWSA       BSS
RNDI1       BSS
RNDI2       BSS
RNDI3       BSS

1 RNDBS
RNDBS
RNDBS
SEQBS
SEQBS
SEQBS
13
RNDIS
RNDIS
RNDIS

DUMMY RANDOM WSA
RANDOM BUFFER 1 FOR FILE 1

PUNCH BUFFER

IO WSA
RANDOM INDEX FOR FILE 1

```

ALLOCATE FOR ALL FETS

```

INPUT      FILEC INBUF, SEQBS, %WSA#IOWSA, IWSAS<
OUTPUT     FILEC OUTBUF, SEQBS, %WSA#IOWSA, IWSAS<
PUNCH      FILEC PRUF, SEQBS, %WSA#IOWSA, IWSAS<
FILE1      RFILEC RNDI1, RNDIS<, %WSA#RNDWSA, RWSAS<
FILE2      RFILEC RNDI2, RNDIS<, %WSA#RNDWSA, RWSAS<
FILE3      RFILEC RNDI3, RNDIS<, %WSA#RNDWSA, RWSAS<

```

VARIABLES USED WITHIN MONITOR

```

OPENF      BSSZ      1
CREATEF     BSSZ      1
RECNO      BSS      1
MSGAB      VFD      60/10HUSER ABEND
           BSS      1

FLAG IS SET IF FILE IS OPEN
FLAG IS SET IF FILE IS CREATED
RANDOM FILE RECORD NUMBER

```

MACRO DEFINITIONS USED BY MONITOR

```

CKEOI      MACRO      FET
           SA1
           RJ
           CKEOI2

```



```

* PUT
ENDM
MACRO FILE OPEN
RJ PUTOPEN
RJ SETWSA
RJ WRITOUT
RJ WRITER
RJ SA2
RJ CKERRS
RJ PUT
ENDM

* READFL
MACRO FILE
LOCAL YESM1
SX7 B3
SA7 RECNO
RJ CKMADESM1
PL XD, YESM1 BE CREATED
NO THE FILE MUST BE CREATED
RJ CKBUFS
SA5 FILE&5
RJ SETWSAS
SA5 FILE&7
RJ SETILN
RJ OPEN
RJ FILE, ALTER, 1
RJ EQ
RJ SETFLAGS
RJ FILE DONEEN CREATED
* YESM1
RJ FILE HAS BEEN
RJ CKEOI
SA2 FILE&5
RJ SETWSA
RJ READIN FILE, RECNO
SA2 FILE
RJ CKERRS
RJ READFL
ENDM

** WRTFL
MACRO FILE
SX7 B3
SA7 RECNO
SA2 FILE&5
RJ SETWSA
RJ WRITOUT FILE, RECNO
SA2 FILE
RJ CKERRS
RJ WRITER
RJ FILE, 1
RJ WRTFL
ENDM

IS FILE OPEN
RESET WSA TO LOC

LOOK FOR WRITE ERRORS

HAS FILE BEEN CREATED

CHECK BUFFER SIZE

INSERT A NEW WSA SIZE

INSERT A NEW INDEX LENGTH

NOTE THAT FILE IS CREATED AND OPEN
RETURN AS CAN NOT READ EMPTY FILE

CHECK TO ENSURE FILE IS OPEN

RESET THE WSA LOCATION

CHECK STATUS FOR ERRORS

RESET WSA LOCATION IN FET

CHECK STATUS FOR ERRORS

```



```

**
**
**
**
**
XPLMON      INITIAL CALL TO XPLMON STARTS HERE
             OPEN INPUT OUTPUT PUNCH FILES

             BSS
             OPEN 1 INPUT, READ, 1
             CKEOI INPUT
             OPEN OUTPUT, WRITE, 1
             CKEOI OUTPUT
             OPEN PUNCH, WRITE, 1
             CKEOI PUNCH
             SX7 7B
             SA7 OPENF
             RJ PROGRAM

             SET FLAGS TO SHOW ABOVE FILES OPEN
             BRANCH TO PROGRAM

**
**
**
**
**
NORMAL ENDING
EQ XPLMON

CHECK FET DESIGNATED BY X1 FOR STATUS FLAGS

CKEOI2      BSS
            SX0 13700GB
            BX1 X1*X0
            SX0 01000B
            IX1 X1-X0
            SX0 100
            ZR X1, ABEND
            EQ CKEOI2

**
**
**
**
**
ENTRY POINT FROM PROGRAM
BRANCH BY CODES

MONITOR     BSS
            SA1 1
            RJ B7 CKCODE
            SX2 CASETOP
            IX3 X1&X2
            SB1 X3
            JP B1&0
            EQ USERAB
            EQ GETSEQ
            EQ PUTSEQ
            EQ READDF
            EQ WRITEF
            EQ PFADECS

CASETOP     3
            3
            3
            3
            3

LOAD IN CALL CODE POINTED TO BY B7
INSURE THAT CALL IS LT MAXCODE

```



```

SB4      3      B2,B4,CKPUTNO
LT        900
SX0      ABEND
EQ

BADPNO

**      IS OUTPUT FILE OPEN
**      PUTOPEN
BSS      1
SA3      OPENF
RX3      X2*X3
NZ        X3,PUTOPEN
SX0      600
EQ      ABEND

**      **      **      **      **      **      **      **      **      **
CODE3    READ A BINARY FILE
READ A RANDOM FILE  CREATE IT IF NECESSARY
**      **      **      **      **      **      **      **      **      **
READF    SAI      B7&1
SA2      B7&4
SA3      B7&5
SA4      B7&2
SB2      X4
SA5      B7&3
SB3      X5
SB1      1
RJ      CKFNO
JP      B2&CSETOP1

CSETOP1  NO
EQ      READF1
EQ      READF2
EQ      READF3
EQ      READFL
EQ      FILE1
EQ      DONE
EQ      FILE2
EQ      DONE
EQ      FILE3
EQ      DONE

**      **      **      **      **      **      **      **      **      **
CODE 4    WRITE BINARY DATA TO FILE
**      **      **      **      **      **      **      **      **      **
WRITEF    SAI      B7&1
**      **      **      **      **      **      **      **      **      **
FWA LOCATION
FILE SIZE
FILE INDEX SIZE
FILE NUMBER
FILE RECORD NUMBER

CHECK FILE NO

```


FILE NUMBER
 RECORD NUMBER
 CHECK FOR PROPER FILE NUMBER
 CHECK FOR AN OPEN FILE

B7&2
 X2
 B7&3
 X3
 1
 CKFNO
 FOPEN
 B2&CSETOP2
 WRITTEF1
 WRITTEF2
 WRITTEF3

SA2
 SB2
 SA3
 SB3
 SB1
 RJ
 RJ
 JP
 NO
 EQ
 EQ
 EQ

CSETOP2

FILE1
 DONE
 FILE2
 DONE
 FILE3
 DONE

WRTFL
 EQ
 WRTFL
 EQ
 WRTFL
 EQ

WRITEF1
 WRITEF2
 WRITEF3

 CODE 5 READ FROM ECS A 26 WORD BLOCK

CORE LOCATION
 ECS LOCATION

B7&1
 X1
 B7&2
 X1
 CLEAN
 B2&26
 ECSE1
 DONE
 555
 ABEND

SA1
 SA1
 SA1
 BXJ
 RE
 EQ
 EQ
 SX0
 EQ

READECS
 ECSE1

 CODE 6 WRITE 26 WORDS FROM CORE TO ECS

CORE LOCATION
 ECS LOCATION

B7&1
 X1
 B7&2
 X1
 B2+26

SA1
 SA1
 SA1
 BXJ
 WE

WRITECS

8

COMMON RANDOM FILE ROUTINES USED BY READFL AND WRTFL MACROS

CHECK FILE NUMBER MUST BE GT 0 AND LT 4

CKFNO BSS 1
LT B2,B0,BADFNO
SB4 4
LT B2,B4,CKFNO
SX0 2100
EQ ABEND

CHECK IF FILE B2 IS CREATED

CKMADE BSS 1
SA4 CREATF
SX0 B1
LX0 B2,X0
BX0 X0*X4
EQ CKMADE

FLAGS FOR CREATED FILES

ENSURE THAT BUFSIZE IN X2 IS LT 3600

CKBUFS BSS 1
SX0 3600
TX0 X2-X0
NG X0,TOOBIG
EQ CKBUFS
SX0 1900
EQ ABEND

IS BUFSIZE LT 3600

QUIT IF NOT LT 3600

INSERT NEW WSA SIZE FROM X2 INTO FET

SETWSAS BSS 1
MX0 18
BX7 X0*X2
BX0 -X0
BX5 X0*X5
BX7 X7&X5
SA7 A5
EQ SETWSAS

CHECK AND INSERT NEW INDEX LENGTH FROM X3

SETILN BSS 1
SX0 100
IX0 X3-X0
NG X0,TOOBIG

INDEX SIZE MUST BE LT 100

MX0	CLEAN UP NEW INDLN
BX3	
LX0	
BX0	
BX7	
LX3	
BX7	
SA7	
EQ	
12	
X0*X3	
18	
-X0	
X5*X0	
18	
X3&X7	
A5	
SETILN	

*** SET OPEN AND CREATED FLAGS FOR A FILE

SETFLAGS BSS 1 37000B
SX0

*** CHECK FOR EOI

BX5	X5*X0
SX0	100CB
IX5	X5-X0
NZ	X5,SETCR
SX0	2100
EQ	ABEND

*** SET FILE CREATED FLAG

SA4	CREATF
SX0	B1
LX0	B2,X0
BX7	X4&X0
SA7	A4

*** SET FILE OPEN FLAG

SA4	OPENF
LX0	3
BX7	X4&X0
SA7	A4
EQ	SETFLAGS

*** CHECK FILE OPEN FLAG

FOPEN	FORMAT	F3	F2	F1	PV	OUT	IN
-------	--------	----	----	----	----	-----	----

FOPEN

BSS	1	OPENF
SA4	18	
SX0	B2&3	
SB3	B3,X0	
LX0	X0*X4	
BX0	X0,FOPEN	
NZ	2000	
SX0		

/*

THE XCOM-ONE COMPILER
AN XPL COMPILER FOR THE CONTROL DATA 6500 COMPUTER

WRITTEN BY
RONALD C. SMEDER
USN POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA
JUNE 1971

THIS IS A LISTING OF VERSION 4 OF XCOM-ONE
DATED 7 JUNE, 1971.

THE XCOM-ONE COMPILER IS DESIGNED TO COMPILE
A 48 CHARACTER SET XPL LANGUAGE INTO CONTROL DATA
6500 COMPASS ASSEMBLER STATEMENTS. AND A DISCUSSION
OF THE HISTORY OF THIS COMPILER. THE ALGORITHMS USED WITHIN IT CAN BE FOUND IN
THE THESIS ENTITLED :

" AN INVESTIGATION INTO COMPILER GENERATION BY
BOOTSTRAPPING TECHNIQUES "
RONALD C. SMEDER
USNPGSCOL
JUNE 1971

THE XCOM-ONE COMPILER IS WRITTEN IN THE XPL
LANGUAGE DESIGNED BY W. M. MCKEEMAN, STANDFORD
UNIVERSITY. FOR FURTHER INFORMATION ON THE XPL
LANGUAGE OR COMPILER GENERATION SYSTEM CONSULT
" A COMPILER GENERATOR " BY MCKEEMAN, AND
OTHERS, PRENTICE-HALL, 1970.

B N F X P L - 6 5 0 0 S E N T A T I O N A L L Y
O F

*/

```
/*  
**  
**<PROGRAM> ::= <STATEMENT LIST> */  
**<STATEMENT LIST> ::= <STATEMENT> */  
**<STATEMENT LIST> ::= <STATEMENT LIST> <STATEMENT> */  
**<STATEMENT> ::= <BASIC STATEMENT> */
```


[illegible]

```

/*
THE FOLLOWING CARDS ARE PRODUCED BY THE XPL-360 SYNTAX
PRE-PROCESSOR, "ANALYZER". THEY ARE USED TO DRIVE THE
PARSING ALGORITHM WITHIN THIS COMPILER.
*/

```

[illegible]

[illegible]

102


```

/* VARIABLES ADDED TO SCANNER TO ALTER PARSING OF STRINGS */
DECLARE(LAST_TOKEN, LAST_LAST_TOKEN, COMMA_TOKEN, PERIOD_TOKEN) FIXED ;

/* VARIABLES ADDED TO SCANNER TO PROVIDE FOR BIT STRINGS */
DECLARE(JBASE, BASE) FIXED ;
DECLARE(LSTRNGM_CHARACTER, INITIAL('STRING LENGTH')) ;
DECLARE(CH, STRING) FIXED ;

/* VARIABLES ADDED TO SCANNER FOR MACRO DEFINITIONS */
DECLARE BALANCE_CHARACTER, LB_FIXED ;
DECLARE MACRO_LIMIT_FIXED, INITIAL(40), MACRO_NAME(40) CHARACTER,
MACRO_TEXT(40) CHARACTER, MACRO_INDEX(256) BIT(8),
TOP_MACRO_FIXED, INITIAL('FFFFFFFF');
DECLARE EXPANSION_COUNT_FIXED, EXPANSION_LIMIT_LITERALLY '300';

/* CONSTANT PROPAGATION VARIABLES USED BY XCOM - ONE ARE : */
DECLARE MAXCNST_LITERALLY '100' ; /* MAX CONST TABLE SIZE */
DECLARE NBRCNST_FIXED ; /* LENGTH OF CNST TABLE */
DECLARE CNST(MAXCNST) FIXED ; /* NUMERIC VALUE */
DECLARE LOCCNST(MAXCNST) FIXED ; /* LABEL LOCATION */

/* SEPARATE STACK NEEDED FOR DECLARATION LISTS AND CASE STMTS */
DECLARE BCNT_FIXED ; /* LENGTH OF BLOC STACK */
DECLARE BLOC(200) FIXED ; /* BRANCH LOCATION STACK */

/* VARIABLES FOR EBCDIC TO BCD CONVERSION */
DECLARE CONVERT_MAP(256) FIXED ;

/* COMPASS ASSEMBLER FILE MAINTENANCE VARIABLES */
DECLARE PROGRAM_START_FIXED ;
DECLARE CODEFILE_LITERALLY '1' ;
DECLARE DATAFILE_LITERALLY '0' ;

/* ADDITIONAL CONTROL TOGGLES USED BY XCOM - ONE :
F ::= FILE CODE
G ::= LIST LOAD MODULE
H ::= CONVERT CHARACTERS TO CDC INTERMEDIATES
I ::= LIST THE CONVERTED CODE
J ::= PUNCH THE CONVERTED CODE
W ::= SYTDUMP AND PRODUCTION NUMBERS PRINTED
*/

/* STATISTIC GATHERING VARIABLES
DECLARE IDCMPARES_FIXED ;
DECLARE INST_FREQ(74) FIXED ;

/* TOTAL TIMES USED */

/* VARIABLES FOR LOCAL PROTECTION
DECLARE ORIGINAL_LITERALLY '0' ;
DECLARE SCRATCH_LITERALLY '01' ;

/* LOCATION NEEDS PROTECTION */
/* DOES NOT NEED PROTECTION */

```



```

/* LIBRARY CALLING VARIABLES, USUALLY DENOTE ENTRY POINT TO A PROC */
DECLARE(CONVERT, MONITOR_CALL, OUTPUT_CALL, CONCATENATE, COMPARE,
INPUT_CALL, NEW_INDEX, SUBSTR_CALL, PUT_SEQ) FIXED;
DECLARE READING_LIBRARY FIXED;
/* TRIGGERS INIT 2 CALL */

/*
P R O C E D U R E S
*/

PAD: PROCEDURE (STRING, WIDTH) CHARACTER;
DECLARE STRING CHARACTER, (WIDTH, L) FIXED;
L = LENGTH(STRING);
IF L >= WIDTH THEN RETURN STRING;
ELSE RETURN STRING || SUBSTR(X70, 0, WIDTH-L);
END PAD;

I_FORMAT: PROCEDURE (NUMBER, WIDTH) CHARACTER;
DECLARE (NUMBER, WIDTH, L) FIXED, STRING CHARACTER;
STRING = NUMBER;
L = LENGTH(STRING);
IF L >= WIDTH THEN RETURN STRING;
ELSE RETURN SUBSTR(X70, 0, WIDTH-L) || STRING;
END I_FORMAT;

ERROR: PROCEDURE (MSG, SEVERITY);
/* PRINTS AND ACCOUNTS FOR ALL ERROR MESSAGES */
/* IF SEVERITY IS NOT SUPPLIED, C IS ASSUMED */
DECLARE MSG CHARACTER, SEVERITY FIXED;
ERROR_COUNT = ERROR_COUNT + 1;
/* IF LISTING IS SUPPRESSED, FORCE PRINTING OF THIS LINE */
IF CONTROL(BYTE('L')) THEN
  OUTPUT = I_FORMAT(CARD_COUNT, 4) || ' ' || BUFFER || ' ';
  OUTPUT = SUBSTR(POINTER, TEXT_LIMIT-CP+MARGIN_CHOP);
  OUTPUT = '*** ERROR: ' || MSG || '
  ' || LAST_PREVIOUS_ERROR WAS DETECTED ON LINE ' ||
  PREVIOUS_ERROR || '***';
  PREVIOUS_ERROR = CARD_COUNT;
  IF SEVERITY > 0 THEN
    IF SEVERITY > 25 THEN
      DO;
        OUTPUT = '*** TOO MANY SEVERE ERRORS, CHECKING ABORTED ***';
      END;
    END IF;
  END IF;
END ERROR;

```



```

        COMPILING = FALSE;
    END; SEVERE_ERRORS = SEVERE_ERRORS + 1;
END ERROR;

CONVERT_CARD :
PROCEDURE (CARD) CHARACTER ; /* 360 48 TO CDC 48 */
DECLARE (CARD,CARD2,CH) CHARACTER ;
DECLARE (I,J) FIXED ;
DO I = 0 TO 79;
J = CONVERT_MAP(BYTE(CARD,I));
DO CASE J;
/* 0 */ CH = '&' ; /* INTERMEDIATE FOR + */
/* 1 */ CH = '%' ; /* INTERMEDIATE FOR ( */
/* 2 */ CH = '<' ; /* INTERMEDIATE FOR ) */
/* 3 */ CH = '#' ; /* INTERMEDIATE FOR = */
/* 4 */ CH = ' ' ; /* INTERMEDIATE FOR ' */
END ;
IF J = 0 THEN
DO ;
CARD2 = SUBSTR(CARD,0,I) || CH || SUBSTR(CARD,I+1) ;
CARD = CARD2 ;
END ;
END ;
RETURN (CARD) ;
END CONVERT_CARD ;

OPTIONS :
PROCEDURE (CARD) ;
DECLARE (CARD,CARD2) CHARACTER ;
IF CONTROL(BYTE('H')) THEN /* CONVERT */
DO ;
CARD2 = CONVERT_CARD(CARD) ; /* PRINT CONVERTED LOAD */
IF CONTROL(BYTE('I')) THEN
OUTPUT = CARD2 ;
IF CONTROL(BYTE('J')) THEN /* PUNCH CONVERTED LOAD */
OUTPUT(2) = CARD2 ;
END ;
END OPTIONS ;

```



```

FILE_360 : PROCEDURE(CARD, FILETYPE) ;
DECLARE(CARD, CARD2) CHARACTER ;
DECLARE(FILETYPE, LN) FIXED ;
IF PRESENT_FILE = FILETYPE THEN
DO ;
DO CASE FILETYPE ;
CARD2 = ! USE CODE ;
CARD2 = ! USE DATA ;
END ;
PRESENT_FILE = FILETYPE ;
CALL OPTIONS(CARD2 || SUBSTR(BLANKS,0,38)) ;
END ;
LN = LENGTH(CARD) ;
IF LN = 80 THEN
CARD2 = CARD ;
ELSE
CARD2 = CARD || SUBSTR(BLANKS,0,79-N) ;
CALL OPTIONS(CARD2) ;
END FILE_360 ;

```

/* CODE EMITTING PROCEDURES */

FORMAT : EMIT AN ASSEMBLER STATEMENT IN THIS FORMAT :

```

1 - 9 ::= (BLANK|*,) || LOCATION
10 ::= BLANK
11 - 16 ::= OP_FIELD
17 ::= BLANK
18 - 35 ::= EXPRESSION FIELD */

```

```

PROCEDURE(LABEL1, OP_FIELD, EXPRES, LABEL2) CHARACTER ;
DECLARE(LABEL1, LABEL2, N) FIXED ;
DECLARE(LOCATION, OP_FIELD, EXPRES, CODE) CHARACTER ;
IF LABEL1 = 0 THEN
LOCATION = BLANK ;
ELSE LOCATION = BLANK || LABEL1 ;
IF LABEL2 = 0 THEN
EXPRES = LL || LABEL2 ;
N = LENGTH(LOCATION) ;
CODE = LOCATION || SUBSTR(BLANKS,0,10 - N) ;
N = LENGTH(OP_FIELD) ;
CODE = CODE || OP_FIELD || SUBSTR(BLANKS,0,7 - N) || EXPRES ;

```



```

N = LENGTH(EXPRES) ;
CODE = CODE || SUBSTR(BLANKS,0,36 - N) ;
RETURN(CODE) ;
END FORMAT ;

```

```

EMIT_DATA:
PROCEDURE(LABEL1,OP,FIELD,EXPRES,LABEL2) ;
DECLARE(LABEL1,LABEL2) FIXED ;
DECLARE(OP,FIELD,EXPRES,DATA) CHARACTER ;
DATA = FORMAT(LABEL1,OP,FIELD,EXPRES,LABEL2) ;
IF CONTROL(BYTE('F')) THEN
CALL FILE_360(DATA,DATAFILE) ;
IF CONTROL(BYTE('E')) THEN
OUTPUT = SPACER1 || DATA || DATA || D_COUNT ;
D_COUNT = D_COUNT + 1 ;
END EMIT_DATA ;

```

```

EMIT:
PROCEDURE(LABEL1,OP,FIELD,EXPRES,LABEL2) ;
DECLARE(LABEL1,LABEL2) FIXED ;
DECLARE(OP,FIELD,EXPRES,CODE) CHARACTER ;
CODE = FORMAT(LABEL1,OP,FIELD,EXPRES,LABEL2) ;
IF CONTROL(BYTE('F')) THEN
CALL FILE_360(CODE,CODEFILE) ;
IF CONTROL(BYTE('E')) THEN
OUTPUT = SPACER2 || CODE || C_COUNT ;
C_COUNT = C_COUNT + 1 ;
END EMIT ;

```

```

EMIT_COMPASS:
PROCEDURE(LABEL1,OCTAL_CODE,I,J,K,LABEL2) ;
DECLARE(LABEL1,LABEL2) FIXED ;
DECLARE(I,J,K) FIXED ;
DECLARE(CONST,CHARACTER ;
DECLARE(OCTAL_CODE,CODE,U,V,VARIABLE_TYPE,MNEO) FIXED ;
DECLARE(CP_CODE,VARIABLE_TYPE,MNEO) FIXED ;
CODE = CP_MAP_1(OCTAL_CODE) ;
IF CODE = 0 & OCTAL_CODE = 0 THEN
DO ;
IF OCTAL_CODE = 1 THEN CODE = I+2 ; ELSE CODE = I+6 ;
END ;
MNEO = CP_MAP_2(CODE) ;
INST_FREQ(CODE) = INST_FREQ(CODE) + 1 ;
CP_CODE = SUBSTR(CP_CODES,MNEO,2) ;

```



```

IF(MNEO > 49) & (MNEO < 99) THEN CP_CODE = CP_CODE || I ;
IF(CODE > 21) & (CODE < 26) THEN
DO; U=K; V=J; END;
ELSE
DO; U=J; V=K; END;
VARIABLE_TYPE = CP_MAP_3(CODE);
CONST = BLANK ;
IF LABEL2 < 0 THEN
DO; LABEL2 = -LABEL2; CONST = LABEL2 ; END ;
ELSE DO;
IF LABEL2 > 0 THEN CONST = LL || LABEL2;
END;
DO CASE VARIABLE_TYPE;
/** 0 **/ ; VAR_FIELD = BLANK ;
/** 1 **/ ; VAR_FIELD = B ;
/** 2 **/ ; VAR_FIELD = X ;
/** 3 **/ ; VAR_FIELD = B ;
/** 4 **/ ; VAR_FIELD = X ;
/** 5 **/ ; VAR_FIELD = X ;
/** 6 **/ ; VAR_FIELD = X ;
/** 7 **/ ; VAR_FIELD = X ;
/** 8 **/ ; VAR_FIELD = X ;
/** 9 **/ ; VAR_FIELD = X ;
/** 10 **/ ; VAR_FIELD = J ;
/** 11 **/ ; VAR_FIELD = B ;
/** 12 **/ ; VAR_FIELD = X ;
/** 13 **/ ; VAR_FIELD = A ;
/** 14 **/ ; VAR_FIELD = B ;
/** 15 **/ ; VAR_FIELD = X ;
/** 16 **/ ; VAR_FIELD = X ;
/** 17 **/ ; VAR_FIELD = A ;
/** 18 **/ ; VAR_FIELD = A ;
/** 19 **/ ; VAR_FIELD = B ;
/** 20 **/ ; VAR_FIELD = B ;
END;
IF (CODE > 21) & (CODE < 26) THEN
IF (VAR_FIELD = MINUS || VAR_FIELD) ;
CALL EMIT(LABEL1, CP_CODE, VAR_FIELD, 0) ;
END EMIT_COMPASS ;

EMIT_CONSTANT :
PROCEDURE(J) FIXED ;
DECLARE(J, I, L) FIXED ;
/** J IS A CONSTANT, RETURN A LOCATION CONTAINING IT */
/** CHECK TO SEE IF CONST IS ALREADY EMITTED */
L = 0 ;
DO I = 0 TO NBRCONST ;

```



```

IF J = CNST(I) THEN L = I ;
END ;
IF L = 0 THEN
RETURN(LOCCNST(L)) ;
ELSE
/* EMIT A CONST AND RECORD LOCATION */
DO ;
LCNT = LCNT + 1 ;
FIELD = VFD_START || FIXV(MP) ;
CALL EMIT_DATA(LCNT, VFD, FIELD, 0) ;
IF NBRCNST < MAXCNST THEN
DO ;
NBRCNST = NBRCNST + 1 ;
CNST(NBRCNST) = J ;
LOCCNST(NBRCNST) = LCNT ;
INFORMATION = INFORMATION || LL || LCNT || BLANK || EQ
|| BLANK || FIXV(MP) ;
END ;
ELSE CALL ERROR('CONSTANT TABLE OVERFLOW', 0) ;
RETURN(LCNT) ;
END ;
END EMIT_CONSTANT ;

```

/* SYMBOL TABLE PROCEDURES */

```

SYTDUMP : PROCEDURE ;
DECLARE I FIXED ;
IF CONTROL(BYTE('W')) THEN
DO ;
OUTPUT = ' SYTDUMP : ID | LOC | TYPE ' ;
DO I = 0 TO SYTLN ;
OUTPUT = ' || SYTID(I) || BLANK || SYTLOC(I) || BLANK
|| SYTTYPE(I) ;
END ;
END ;
END SYTDUMP ;

```

```

ENTER1 : PROCEDURE(ID, IDLOC, IDTYPE) ;
DECLARE(IDLOC, IDTYPE) FIXED ;
DECLARE ID CHARACTER ;
SYTLN = SYTLN + 1 ;
SYTID(SYTLN) = ID ;
SYTLOC(SYTLN) = IDLOC ;

```



```

SYTTYPE(SYTLN) = IDTYPE ;
CALL SYTDUMP ;
END ENTER1 ;

ENTER3 :
PROCEDURE(ID, IDLOC, IDTYPE) FIXED ;
DECLARE(IDLOC, IDTYPE, I) FIXED ;
DECLARE ID CHARACTER ;
IF IDTYPE = PROC THEN
DO ;
CALL ENTER1(ID, IDLOC, IDTYPE) ;
DEPTH = DEPTH + 1 ;
PROC_MARK(DEPTH) = SYTLN ;
LAST_PARAM(DEPTH) = SYTLN ;
RETURN(SYTLN) ;
END ;
IF IDTYPE = PARAM THEN
DO ;
CALL ENTER1(ID, IDLOC, IDTYPE) ;
LAST_PARAM(DEPTH) = SYTLN ;
RETURN(SYTLN) ;
END ;
IF SYTLN > 0 THEN
DO I = PROC_MARK(DEPTH) TO SYTLN ;
IF ID = SYTID(I) THEN
DO ;
IF SYTTYPE(I) = PARAM THEN
DO ;
IF IDTYPE = UNKNOWN THEN
SYTTYPE(I) = IDTYPE ;
RETURN(I) ;
END ;
ELSE
DO ;
CALL ERROR('MULTIPLE DECLARATION', 0) ;
RETURN(I) ;
END ;
END ;
END ;
CALL ENTER1(ID, IDLOC, IDTYPE) ;
RETURN(SYTLN) ;
END ENTER3 ;

SYTSEARCH :
PROCEDURE(ID) FIXED ;
DECLARE ID CHARACTER ;

```



```

DECLARE I FIXED ;
I = SYTLN ;
DO WHILE I > 0 ;
IDCOMPARES = IDCOMPARES + 1 ;
IF SYTID(I) = ID THEN
DO ;
REFER(I) = REFER(I) + 1 ;
RETURN(I) ;
END ;
I = I - 1 ;
END ;
CALL ERROR('UNDECLARED ' || ID , 0) ;
LCNT = LCNT + 1 ;
I = ENTER3(ID,LCNT,FIXEDTYPE) ;
CALL EMIT_DATA(LCNT,BSS,ONE,0) ;
RETURN(I) ;
END SYTSEARCH ;

```

```

CUTBACK : PROCEDURE ;
/* WIPE OUT ALL LOCAL VARIABLES IN PROC */
SYTLN = LAST_PARAM(DEPTH) ;
DO I = PROC_MARK(DEPTH) + 1 TO SYTLN ;
SYTID(I) = BLANK ;
END ;
DEPTH = DEPTH - 1 ;
IF DEPTH < 0 THEN
DO ;
CALL ERROR('TOO MANY CUTBACKS',1) ;
DEPTH = 0 ;
END ;
CALL SYTDUMP ;
END CUTBACK ;

```

```

/* NUMERIC VARIABLE PROCEDURES */

```

```

PROTECT : PROCEDURE(PTR) FIXED ;
/* PTR IS BEST BET FOR A FREE LOCATION, IF IT ISNT FREE
THEN CREATE A NEW LOCATION AND THEN STORE */
DECLARE(PTR,1) FIXED ;
IF FIXO(PTR) = ORIGINAL THEN

```



```

DO ; /* FORCED TO CREATE NEW LOCATION */
LCNT = LCNT + 1 ;
CALL EMIT_DATA(LCNT,BSS,ONE,0) ;
I = LCNT ;
FIXO(MP) = SCRATCH ;
END ;
ELSE I = FIXV(PTR) ;
CALL EMIT_COMPASS(0,51,7,0,0,I) ; /* STORE X7 */
RETURN(I) ;
END PROTECT ;

NOT_STRING :
PROCEDURE FIXED ;
DECLARE(J,K) FIXED ;
J = TYPE(MP) ;
K = TYPE(SP) ;
IF J = CHRTYPE | K = CHRTYPE THEN
DO ;
CALL ERROR('ILLEGAL STRING OPERATION',1) ;
RETURN(0) ;
END ;
ELSE
RETURN(1) ;
END NOT_STRING ;

FLOAT_OPS : /* MULTIPLY DIVIDE OPERATIONS */
PROCEDURE(CODE) ;
DECLARE(CODE,J,I) FIXED ;
IF NOT_STRING THEN
DO ;
J = FIXV(MP) ;
I = FIXV(SP) ;
CALL EMIT_COMPASS(0,51,1,0,0,I) ; /* SA1 <I> */
CALL EMIT_COMPASS(0,27,2,0,0,1,0) ; /* PX2 BC,X1 */
CALL EMIT_COMPASS(0,24,1,0,2,0,0) ; /* NX1 80,X2 */
CALL EMIT_COMPASS(3,51,3,0,0,J) ; /* SA3 <J> */
CALL EMIT_COMPASS(0,27,4,0,3,0,0) ; /* PX4 80,X3 */
CALL EMIT_COMPASS(0,24,3,0,4,0,0) ; /* NX3 80,X4 */
CALL EMIT_COMPASS(0,24,3,0,3,0,0) ; /* FX5 X3 X1 */
CALL EMIT_COMPASS(0,26,6,2,5,0,0) ; /* UX6 82,X5 */
CALL EMIT_COMPASS(0,23,7,2,6,0,0) ; /* AX7 82,X6 */
FIXV(MP) = PROTECT(MP) ; /* PROTECT X7 WITH NEW LOCATION */
END ;
END FLOAT_OPS ;

```



```

FIXED_OPS : /* ADD SUBTRACT OPERATIONS */
PROCEDURE(CODE) ;
DECLARE(CODE,J,I) FIXED ;
IF NOT_STRING THEN
DO ;
J = FIXV(MP) ;
I = FIXV(SP) ;
CALL EMIT_COMPASS(0,51,1,0,0,I) ; /* SA1 <I> */
CALL EMIT_COMPASS(0,51,2,0,0,J) ; /* SA2 <J> */
CALL EMIT_COMPASS(0,CODE,7,2,1,0) ; /* IX7 X2 X1 */
FIXV(MP) = PROTECT(MP) ; /* PROTECT X7 WITH NEW LOCATION */
END ;
FIXED_OPS ;

LOGIC_OPS : /* AND OR OPERATIONS */
PROCEDURE(CODE) ;
DECLARE(CODE,J,I) FIXED ;
IF NOT_STRING THEN
DO ;
J = FIXV(MP) ;
I = FIXV(SP) ;
CALL EMIT_COMPASS(0,51,1,0,0,I) ; /* SA1 <I> */
CALL EMIT_COMPASS(0,51,2,0,0,J) ; /* SA2 <J> */
CALL EMIT_COMPASS(0,CODE,7,1,2,0) ; /* BX7 X1,X2 */
FIXV(MP) = PROTECT(MP) ; /* PROTECT X7 WITH NEW LOCATION */
END ;
LOGIC_OPS ;

MAKE_STRING :
PROCEDURE(PTR,INDEX) FIXED ;
DECLARE(PTR,COMPASS(0,51,0,0,0,0),FIXV(PTR)) ; /* LOAD VALUE */
CALL EMIT_COMPASS(0,51,0,0,0,0) ; /* BUILD A CALL TO CONVERT TO STRING */
IF TYPE(PTR) = CHRTYPE THEN
DO ;
CALL EMIT_COMPASS(0,51,7,0,0,0) ; /* BX7 X1 */
CALL EMIT_COMPASS(0,51,7,0,0,0) ; /* STORE VALUE */
INDEX = INDEX + 1 ;
CALL EMIT_COMPASS(0,71,7,0,0,-INDEX) ; /* SX7 INDEX */
CALL EMIT_COMPASS(0,51,7,0,0,0) ; /* STORE INDEX */
CALL EMIT_COMPASS(0,1,0,0,0,0) ; /* RJ */
TYPE(PTR) = CHRTYPE ;
END ;
FIXV(PTR) = PROTECT(PTR) ; /* PROTECT DESCRIPTOR */
END MAKE_STRING ;

```



```

ASSIGN:
PROCEDURE (VAR, EXP) FIXED ;
DECLARE(MP) = CHRTYPE THEN CALL MAKE_STRING(SP, BUFFER1) ;
IF TYPE = FIXV(MP) ;
VAR = FIXV(SP) ;
EXP = FIXV(MP) ;
CALL EMIT_COMPASS(0,51,1,0,0,EXP) ; /* LOAD EXP */
CALL EMIT_COMPASS(0,10,7,1,0,0) ; /* BX7 X1 */
IF VAR = OUTPUT_CALL THEN
DO ;
CALL EMIT_COMPASS(0,51,7,0,0,OUTPUT_CALL+3) ;
CALL EMIT(0,RJ,BLANK,OUTPUT_CALL) ;
RETURN ;
END ;
IF FIXL(MP) < 0 THEN
DO ; /* INDIRECT ADDRESS, LOAD AGAIN */
CALL EMIT_COMPASS(0,51,2,0,0,VAR) ; /* SA2 <VAR> */
CALL EMIT_COMPASS(0,53,7,2,0,0) ; /* SA7 X2+BC */
RETURN ;
END ;
CALL EMIT_COMPASS(0,51,7,0,0,VAR) ; /* SA7 <VAR> */
END ASSIGN ;

/*
TIME AND DATE

PRINT TIME:
PROCEDURE (MESSAGE, CHARACTER, T) FIXED ;
DECLARE MESSAGE = MESSAGE || T/360000 || ',' || T MOD 360000 / 6000 || ',' ;
MESSAGE = MESSAGE / 100 || ',' ;
T = T MOD 100 ; /* MESSAGE = MESSAGE || '0' ;
IF T < 10 THEN MESSAGE = MESSAGE || '0' ;
OUTPUT = MESSAGE || T || ',' ;
END PRINT_TIME ;

PRINT DATE AND TIME:
PROCEDURE (MESSAGE, D, CHARACTER, (D, T, YEAR, DAY, M) FIXED ;
DECLARE MESSAGE = MESSAGE || CHARACTER INITIAL ('JANUARY', 'FEBRUARY', 'MARCH',
APRIL', 'MAY', 'JUNE', 'JULY', 'AUGUST', 'SEPTEMBER', 'OCTOBER',
NOVEMBER', 'DECEMBER') ;

```



```

CONCATENATE = SYTLOC(SYTSEARCH('CONCAT'));
INPUT_CALL = SYTLOC(SYTSEARCH('INPUT'));
NEW_INDEX = SYTLOC(SYTSEARCH('NEW_INDEX'));
SUBSTR_CALL = SYTLOC(SYTSEARCH('SUBSTR'));
PUT_SEQ = SYTLOC(SYTSEARCH('PUT_SEQ'));
END_INITIALIZE2;

```

```

STRING_ADDITION:
PROCEDURE (REQUEST);
DECLARE (REQUEST,DESC) FIXED;
DO CASE REQUEST;

/* 0 * STORE A CONSTANT STRING */
DO;
LCNT = LCNT + 1;
S = VAR(MP);
I = LENGTH(S);
/* VFD THE STRING INTO DATA AREA */
N = 0;
J = 0;
DO WHILE J < I; /* SEND OUT 10 CHARACTERS PER VFD */
K = I - J; THEN K = 0;
IF K > 10 THEN K = 10;
FIELD = VFD_START || SUBSTR(VAR(MP),J,K);
CALL EMIT_DATA(N,VFD,FIELD,0);
N = 0;
J = J + 10;
END;
/* BUILD THE DESCRIPTOR */
INDEX = INDEX + 1; /* ECS INDEX NUMBER */
DESC = SHL(I,18) + (INDEX & "3FFFF");
/* STORE RELOAD INFORMATION ABOUT CONSTANT STRING */
RE-CNT = RE-CNT + 1;
IF RE-CNT > MAXSTRING THEN
DO; /* ESCLOAD ARRAY FULL */
CALL ERROR('TOO MANY CONSTANT STRINGS',1);
RE-CNT = 0;
END;
ECSLOAD(RE-CNT) = SHL(DESC,18) + (LCNT & "3FFFF");
/* SET TYPES AND LOCATION */
FIXV(MP) = DESC;
TYPE(MP) = CHRTYPE;
END;

/* 1 CALL TO COMPARE STRINGS */

```



```

DO ; NOT FIXED THEN
IF SET_PARM(COMPAR) ;
ELSE CALL_ERROR('COMPARING MIXED TYPES',1) ;
TYPE(MP) = FIXEDTYPE ;
END ;

/* 2 CALL CONCATENATE */
DO ;
CALL MAKE_STRING(MP,BUFFER1) ;
CALL MAKE_STRING(SP,BUFFER2) ;
CALL SET_PARM(CONCATENATE) ;
TYPE(MP) = CHRTYPE ;
END ;

END ; /* OF STRING ADDITION CASE STATEMENT */
END STRING_ADDITION ;

/*
CARD IMAGE HANDLING PROCEDURE
*/

GET_CARD: PROCEDURE ;
/* DOES ALL CARD READING AND LISTING */
DECLARE I FIXED, (TEMP, TEMPO, REST) CHARACTER, READING BIT(1) ;
/*
IF BUFFER = INPUT ;
IF LENGTH(BUFFER) = 0 THEN
DO ; /* SIGNAL FOR EOF */
CALL_ERROR ('EOF MISSING OR COMMENT STARTING IN COLUMN 1.',1) ;
BUFFER = PAD (' ' /* */ EOF;END;EOF', 80) ;
END ;
ELSE CARD_COUNT = CARD_COUNT + 1 ; /* USED TO PRINT ON LISTING */
IF MARGIN_CHOP > 0 THEN
DO ; /* THE MARGIN CONTROL FROM DOLLAR | */
I = LENGTH(BUFFER) - MARGIN_CHOP ;
REST = SUBSTR(BUFFER,I) ;
BUFFER = SUBSTR(BUFFER, 0, I) ;
END ;
ELSE REST = '' ;
TEXT = BUFFER ;
TEXT LIMIT = LENGTH(TEXT) - 1 ;
IF CONTROL(BYTE('M')) THEN OUTPUT = BUFFER ;
ELSE IF CONTROL(BYTE('L')) THEN
DO ;

```



```

OUTPUT = I_FORMAT (CARD_COUNT, 4) || ' ' || BUFFER || ' ' ||
C_COUNT + D_COUNT || REST || INFORMATION ;
INFORMATION = BLANK ;
END ;
CP = 0 ;
END GET_CARD ;

```

THE SCANNER PROCEDURES

```
CHAR:
PROCEDURE;
/* USED; FOR STRINGS TO AVOID CARD BOUNDARY PROBLEMS */
CP = CP + 1;
IF CP <= TEXT_LIMIT THEN RETURN;
CALL GET_CARD;
END CHAR;
```

```
DEBLANK:
PROCEDURE;
/* USED BY BCHAR */
CALL CHAR;
DO WHILE BYTE(TEXT, CP) = BYTE(' ');
CALL CHAR;
END;
END DEBLANK;
```

```

BCHAR:
PROCEDURE;
/* USED FOR BIT STRINGS */
DO FOREVER;
  CALL DEBLANK;
  CH = BYTE(TEXT, CP);
  IF CH ^= BYTE(' ') THEN RETURN;
  /* (BASE WIDTH) */
  CALL DEBLANK;
  JBASE = BYTE(TEXT, CP) - "FO"; /* WIDTH */
  IF JBASE < 1 | JBASE > 4 THEN
    DO;
      CALL ERROR ('ILLEGAL BIT STRING WIDTH: ' || SUBSTR(TEXT, CP, 1));
      JBASE = 4; /* DEFAULT WIDTH FOR ERROR */
    END;
  BASE = SHL(1, JBASE);
  CALL DEBLANK;
  IF BYTE(TEXT, CP) ^= BYTE(' ') THEN
    CALL ERROR ('MISSING ' IN BIT STRING', 0);
END;

```



```

END BCHAR;

BUILD_BCD: PROCEDURE (C);
DECLARE C BIT(8);
IF LENGTH(BCD) > 0 THEN
    BCD = BCD || X1;
ELSE
    BCD = SUBSTR(X1 || X1, 1);
END BUILD_BCD;

SCAN: PROCEDURE (S1, S2) FIXED;
DECLARE COUNT(3) = CALLCOUNT(3) + 1;
FAILSOFT = TRUE;
BCD = ''; NUMBER_VALUE = 0;
SCAN1: DO
    FOREVER;
    IF CP > TEXT_LIMIT THEN CALL GET_CARD;
    ELSE
        DO; /* DISCARD LAST SCANNED VALUE */
            TEXT_LIMIT = TEXT_LIMIT - CP;
            TEXT = SUBSTR(TEXT, CP);
            CP = 0;
        END;
        /* BRANCH ON NEXT CHARACTER IN TEXT
        DO CASE CHARTYPE(BYTE(TEXT));
            /* CASE 0 */
            /* ILLEGAL CHARACTERS FALL HERE */
            CALL_ERROR ('ILLEGAL CHARACTER: ' || SUBSTR(TEXT, 0, 1));
            /* CASE 1 */
            /* BLANK */
        DO;
            CP = 1;
            DO WHILE BYTE(TEXT, CP) = BYTE(' ') & CP <= TEXT_LIMIT;
                CP = CP + 1;
            END;
            CP = CP - 1;
        END;
    END;

```



```

/* CASE 2 */
/* STRING QUOTE (.): CHARACTER STRING */
/* KLUDGE TO REMOVE <ID> AND , */
/* IF (LAST_TOKEN = COMMA_TOKEN & LAST_LAST_TOKEN = STRING) |
   LAST_TOKEN = IDENT
   LAST_TOKEN = PERIOD_TOKEN THEN
DO;
  TOKEN = PERIOD_TOKEN;
  CP = 1;
  RETURN;
END;
ELSE
DO FOREVER;
  TOKEN = STRING;
  S1 = 1;
  CP = CP + 1;
  DO WHILE BYTE(TEXT, CP) = BYTE(PERIOD);
    IF CP <= TEXT_LIMIT THEN
      CP = CP + 1;
    ELSE
      DO; /* STRING BROKEN ACROSS CARD BOUNDARY */
        IF LENGTH(BCD) + CP > 257 THEN
          CALL ERROR(LSTRNGM, 0);
          RETURN;
        END;
        IF CP > S1 THEN
          BCD = BCD || SUBSTR(TEXT, S1, CP - S1);
          TEXT = X1;
          CP = 0;
          CALL GET_CARD;
        S1 = 0;
      END;
    END;
    IF LENGTH(BCD) + CP > 257 THEN
      DO;
        CALL ERROR(LSTRNGM, 0);
        RETURN;
      END;
    IF CP > S1 THEN
      BCD = BCD || SUBSTR(TEXT, S1, CP - S1);
      CALL CHAR;
      IF BYTE(TEXT, CP) = BYTE(PERIOD) THEN
        RETURN;
      IF LENGTH(BCD) > 255 THEN
        DO;
          CALL ERROR(LSTRNGM, 0);

```



```

RETURN ;
END ;
BCD = BCD ;
TEXT_LIMIT = TEXT_LIMIT - CP ;
TEXT = SUBSTR(TEXT, CP) ;
CP = 0 ;
END ;

/* CASE 3 */
DO ;
  /* BIT QUOTE($): BIT STRING */
  JBASE = 4 ;
  BASE = SHL(1, JBASE) ;
  /* DEFAULT WIDTH */
  TOKEN = NUMBER ;
  /* ASSUME SHORT BIT STRING */
  S1 = 0 ;
  CALL BCHAR ;
  DO WHILE CH = BYTE(DOLLAR) ;
    S1 = S1 + JBASE ;
    IF CH >= "FO" THEN S2 = CH - "FO" ; /* DIGITS */
    ELSE S2 = CH - "B7" ; /* LETTERS */
    IF S2 >= BASE - 1 S2 < 0 THEN
      CALL ERROR('ILLEGAL CHARACTER IN BIT STRING: '
        || SUBSTR(TEXT, CP, 1)) ;
    IF S1 > 32 THEN TOKEN = STRING ; /* LONG BIT STRING */
    IF TOKEN = STRING THEN
      DO WHILE S1 - JBASE >= 8 ;
        IF LENGTH(BCD) > "FF" THEN
          DO ;
            CALL ERROR (LSTRNGM, 0) ;
            RETURN ;
          END ;
          S1 = S1 - 8 ;
          CALL BUILD_BCD (SHR(NUMBER_VALUE, S1-JBASE)) ;
        END ;
        VALUE = SHL(NUMBER_VALUE, JBASE) + S2 ;
        CALL BCHAR ;
        /* OF DO WHILE CH... */
      END ;
      CP = CP + 1 ;
      IF TOKEN = STRING THEN
        IF LENGTH(BCD) > "FF" THEN CALL ERROR (LSTRNGM, 0) ;
        ELSE CALL BUILD_BCD (SHL(NUMBER_VALUE, 8 - S1)) ;
      END ;
    END ;
  /* CASE 4 */
  DO FOREVER ; /* A LETTER: IDENTIFIERS AND RESERVED WORDS */
    DO CP = CP + 1 TO TEXT_LIMIT ;
      IF NOT_LETTER_OR_DIGIT(BYTE(TEXT, CP)) THEN

```



```

DO; /* END OF IDENTIFIER */
  IF CP > 0 THEN BCD = BCD || SUBSTR(TEXT, 0, CP);
  S1 = LENGTH(BCD);
  IF S1 > 1 THEN IF S1 <= RESERVED_LIMIT THEN
    /* CHECK FOR RESERVED_WORDS #7
    DO I = V_INDEX(S1-1) TO V_INDEX(S1) - 1;
      IF BCD = V(I) THEN
        DO;
          TOKEN = I;
          RETURN;
        END;
      END;
    END;
  DO I = MACRO_INDEX(S1-1) TO MACRO_INDEX(S1) - 1;
    IF BCD = MACRO_NAME(I) THEN
      DO;
        BCD = MACRO_TEXT(I);
        IF EXPANSION_COUNT < EXPANSION_LIMIT THEN
          EXPANSION_COUNT = EXPANSION_COUNT + 1;
        ELSE OUTPUT =
          '** WARNING, TOO MANY EXPANSIONS FOR ' ||
          MACRO_NAME(I) || ' LITERALLY: ' || BCD;
        TEXT = SUBSTR(TEXT, CP);
        TEXT_LIMIT = TEXT_LIMIT - CP;
        IF LENGTH(BCD) + TEXT_LIMIT > 255 THEN
          DO;
            IF LB + TEXT_LIMIT > 255 THEN
              CALL ERROR7('MACRO EXPANSION TOO LONG');
            ELSE
              DO;
                BALANCE = TEXT || BALANCE;
                LB = LENGTH(BALANCE);
                TEXT = BCD;
              END;
            END;
          END;
        ELSE TEXT = BCD || TEXT;
        BCD = '';
        TEXT_LIMIT = LENGTH(TEXT) - 1;
        CP = 0;
        GO TO SCAN1;
      END;
    END;
  END;
  /* RESERVED_WORDS EXIT HIGHER: THEREFORE <IDENTIFIER> */
  TOKEN = IDENT;
  RETURN;
END;

END; /* END OF CARD */
BCD = BCD || TEXT;

```



```

CALL GET_CARD;
CP = -1;
END;

/* CASE 5 */
DO; /* DIGIT: A NUMBER */
  TOKEN = NUMBER;
  DO FOREVER;
    DO CP = CP TO TEXT_LIMIT;
      S1 = BYTE(TEXT, CP);
      IF S1 < "FO" THEN RETURN;
      NUMBER_VALUE = IC*NUMBER_VALUE + S1 - "FO";
    END;
    CALL GET_CARD;
  END;
END;

/* CASE 6 */
DO; /* A /: MAY BE DIVIDE OR START OF COMMENT */
  CALL CHAR;
  IF BYTE(TEXT, CP) /= BYTE('/') THEN
    DO;
      TOKEN = DIVIDE;
      RETURN;
    END;
  /* WE HAVE A COMMENT */
  S1, S2 = BYTE(' ');
  DO WHILE S1 /= BYTE('$');
    IF S1 = BYTE('$') THEN CHARACTER /*
      DO; /* A CONTROL CHARACTER */
        CONTROL(S2) = CONTROL(S2);
        IF S2 = BYTE('Z') & READING_LIBRARY THEN
          DO;
            READING_LIBRARY = FALSE;
            CALL INITIALIZE2;
          END;
        ELSE
          IF S2 = BYTE('T') THEN CALL TRACE;
          ELSE IF S2 = BYTE('U') THEN CALL UNTRACE;
          ELSE IF S2 = BYTE('I') THEN
            IF CONTROL(S2) THEN
              MARGIN_CHOP = TEXT_LIMIT - CP + 1;
            ELSE
              MARGIN_CHOP = 0;
          END;
        END;
      END;
    END;
  END;

```



```

S1 = S2;
CALL CHAR;
S2 = BYTE(TEXT, CP);
END;
END;

/* CASE 7 */ SPECIAL CHARACTERS */
DO; TOKEN = TX(BYTE(TEXT));
  CP = 1;
  RETURN;
END;

/* CASE 8 */
; /* NOT USED IN SKELETON (BUT USED IN XCOM) */

END; /* OF CASE ON CHARTYPE */
CP = CP + 1; /* ADVANCE SCANNER AND RESUME SEARCH FOR TOKEN */
END;
END SCAN;

/*
INITIALIZATION

EMIT BUILT-IN CODE ;
/* PROC TO WRITE LOAD FILE AND PROCEDURES SHL, SHR, GETWORD, PUTWORD */
PROCEDURE ;
DECLARE IDS(4) CHARACTER INITIAL('SHL', 'SHR', 'GETWORD', 'PUTWORD');
DECLARE CODES(2) FIXED INITIAL(23, 22);
DECLARE LABELS(5) FIXED ;

/* EMIT INITIAL LOADING STATEMENTS FOR COMPASS */
PRESENT FILE = CODEFILE ;
CALL EMIT(0, 'IDENT', PROGRAM, 0) ;
CALL EMIT(0, 'ENTRY', PROGRAM, 0) ;
CALL EMIT(0, 'EXT', MONITOR, 0) ;
PRESENT FILE = DATAFILE ;
/* SET UP LABEL FOR BRANCH BACK TO TOP OF PROGRAM */
LCNT = LCNT + 1 ;
CALL EMIT(LCNT, NO, BLANK, 0) ;
PROGRAM_START = LCNT ;

/* EMIT BRANCH OVER BUILT IN PROCEDURES */
LCNT = LCNT + 1 ;
CALL EMIT(0, EQ, BLANK, LCNT) ;
N = LCNT ;

/* EMIT CODE FOR SHL AND SHR */

```



```

DO I = 0 TO 1 ;
DO J = 0 TO 4 ;
LCNT = LCNT + 1 ;
LABELS(J) = LCNT ;
LEND ;
ENTERI( IDS(I), LABELS(0), FIXEDPROC) ;
J = ENTERI( LABELS(0), BSS, ONE, 0) ;
CALL EMIT(0, EQ, 0, LABELS(1)) ;
CALL EMIT(0, EQ, 2, 1, 0, 4) ;
DO K = 2 TO 4 ;
CALL EMIT( LABELS(K), BSS, ONE, 0) ;
LEND ;
ENTERI( BLANK, LABELS(K), FIXEDTYPE) ;
LEND ;
ENTERI( LABELS(1), NO, BLANK, 0) ;
CALL EMIT( LABELS(0), 51, 1, 0, 0, LABELS(3)) ;
CALL EMIT( COMPASS(0), 51, 0, 0, LABELS(4)) ;
CALL EMIT( COMPASS(0), 61, 2, 0, 0, LABELS(1)) ;
CALL EMIT( COMPASS(0), CODES(1), 7, 2, 1, 0) ;
CALL EMIT( COMPASS(0), 51, 7, 0, 0, LABELS(2)) ;
CALL EMIT(0, EQ, BLANK, LABELS(0)) ;
END ;

/* EMIT CODE FOR GETWORD AND PUTWORD */

K = FIXEDPROC ;
DO I = 2 TO 3 ;
DO J = 0 TO 3 ;
LCNT = LCNT + 1 ;
LABELS(J) = LCNT ;
LEND ;
ENTERI( IDS(I), LABELS(0), K) ;
J = ENTERI( LABELS(0), BSS, ONE, 0) ;
K = ENTERI( LABELS(0), EQ, BLANK, LABELS(1)) ;
CALL EMIT(0, EQ, BLANK, LABELS(1)) ;
DO J = 2 TO 3 ;
CALL EMIT( LABELS(J), BSS, ONE, 0) ;
LEND ;
ENTERI( BLANK, LABELS(J), FIXEDTYPE) ;
CALL EMIT( LABELS(1), NO, BLANK, 0) ;
IF I = 2 THEN
/* GETWORD CODE */
DO ;
CALL EMIT_ COMPASS(0, 51, 1, 0, 0, LABELS(3)) ;
CALL EMIT_ COMPASS(0, 53, 2, 2, 0, 0) ;
CALL EMIT_ COMPASS(0, 10, 7, 2, 0, 0) ;
CALL EMIT_ COMPASS(0, 51, 7, 0, 0, LABELS(2)) ;
END ;
ELSE
/* PUTWORD CODE */

```



```

DO ;
CALL EMIT_COMPASS(0,51,1,0,0,LABELS(2)) ;
CALL EMIT_COMPASS(0,10,7,1,0,0) ;
CALL EMIT_COMPASS(0,51,7,0,0,LABELS(3)) ;
END ;
CALL EMIT(0,EQ,BLANK,LABELS(0)) ;
END ;
CALL EMIT(N,NO,BLANK,0) ; /* CLOSE BRANCH OVER BUILT IN PROCS */
OUTPUT = '*** BUILT IN CODE EMITTED ***' ;
END EMIT_BUILT_IN_CODE ;

INITIALIZATION:
PROCEDURE ;
EJECT PAGE ;
CALL PRINT DATE_AND_TIME (' XCCM-ONE COMPILER -- USN POSTGRADUATE SCHOOL
VERSION OF 711581,0) ;
DOUBLE SPACE ;
CALL PRINT DATE_AND_TIME ('TODAY IS ', DATE, TIME) ;
DOUBLE SPACE ;
/* INITIALIZE CODE EMISSION VARIABLES */
MULT = SUBSTR(SPECIAL_CHAR,0,1) ;
DIV = SUBSTR(SPECIAL_CHAR,1,1) ;
PLUS = SUBSTR(SPECIAL_CHAR,3,1) ;
MINUS = SUBSTR(SPECIAL_CHAR,2,1) ;
COMMA = SUBSTR(SPECIAL_CHAR,4,1) ;
DOLLAR = SUBSTR(SPECIAL_CHAR,5,1) ;
X = SUBSTR(ALPHABET,23,1) ;
A = SUBSTR(ALPHABET,0,1) ;
B = SUBSTR(ALPHABET,1,1) ;
D = SUBSTR(ALPHABET,3,1) ;
LL = SUBSTR(ALPHABET,11,1) ;
EQ = 'EQ' ;
NO = 'NO' ;
BSS = 'BSS' ;
ONE = '1' ;
RJ = 'RJ' ;
VFD = 'VFD' ;
VFD_START = '60/' ;
FIELD = 'BLANKS' ;
BLANKS = 'BLANKS' ;
SPACER1 = SUBSTR(BLANKS,0,26) ;
SPACER2 = SUBSTR(BLANKS,0,34) ;
PROGRAM = 'PROGRAM' ;
MONITOR = 'MONITOR' ;

```



```

/* INITIALIZE COMPILER VARIABLES */
C_COUNT,D_COUNT,LCNT,SYTLN,BCNT,NBRCNST = 1 ;
DEPTH,IDCOMPARES = 0 ;
PROC_MARK(0) = 1 ;
INFORMATION = BLANK ;
BUFFER1 = 0 ;
BUFFER2 = 1 ;
INDEX = 2 ;
READING_LIBRARY = TRUE ;

/* INIT CONVERSION TABLE USED BY CONVERT */
DO I = 0 TO 255 ;
  CONVERT_MAP(I) = 0 ;
END ;
CONVERT_MAP(BYTE(PLUS)) = 1 ;
CONVERT_MAP(BYTE(',')) = 2 ;
CONVERT_MAP(BYTE('')) = 3 ;
CONVERT_MAP(BYTE('=')) = 4 ;

/* ZERO OUT SYT */
DO I = 0 TO SYTLIMIT-1 ;
  SYTID(I) = '**' ;
  SYTLOC(I),SYTTYPE(I) = 0 ;
  REFER(I) = 0 ;
  DECLARED_ON_LINE(I) = 0 ;
END ;

DO I = 0 TO 73 ;
  INST_FREQ(I) = 0 ;
END ;

DO I = 1 TO NT ;
  S = V(I) ;
  IF S = '<NUMBER>' THEN NUMBER = I ; ELSE
  IF S = '<IDENTIFIER>' THEN IDENT = I ; ELSE
  IF S = '/' THEN DIVIDE = I ; ELSE
  IF S = '!' THEN EOFILE = I ; ELSE
  IF S = '<STRING>' THEN STRING = I ; ELSE
  IF S = COMMA THEN COMMA_TOKEN = I ; ELSE
  IF S = PERIOD THEN
    DO ;
      STOPIT(I) = TRUE ;
      PERIOD_TOKEN = I ;
    END ;
  ELSE

```



```

END; IDENT = NT THEN RESERVED_LIMIT = LENGTH(V(NT-1));
ELSE RESERVED_LIMIT = LENGTH(V(NT));
V(EofILE) = 'EOF';
STOPIT(EofILE) = TRUE;
DO I = 0 TO 255;
  NOT_LETTER_OR_DIGIT(I) = TRUE;
END;
DO I = 0 TO LENGTH(ALPHABET) - 1;
  J = BYTE(ALPHABET, I);
  TX(J) = I;
  NOT_LETTER_OR_DIGIT(J) = FALSE;
  CHARTYPE(J) = 4;
END;
DO I = 0 TO 9;
  J = BYTE('0123456789', I);
  NOT_LETTER_OR_DIGIT(J) = FALSE;
  CHARTYPE(J) = 5;
END;
DO I = V_INDEX(0) TO V_INDEX(1) - 1;
  J = BYTE(V(I));
  TX(J) = I;
  CHARTYPE(J) = 7;
END;
CHARTYPE(BYTE(BLANK)) = 1;
CHARTYPE(BYTE(DOLLAR)) = 3;
CHARTYPE(BYTE(PERIOD)) = 2;
CHARTYPE(BYTE('/')) = 6;

/* FIRST SET UP GLOBAL VARIABLES CONTROLLING SCAN, THEN CALL IT */
CP = 0; TEXT_LIMIT = -1;
TEXT = '';
CCNTROL(BYTE('L')) = TRUE;
CALL EMIT_BUILT_IN_CODE;
CALL SCAN;

/* INITIALIZE THE PARSE STACK */
SP = 1; PARSE_STACK(SP) = EofILE;

END INITIALIZATION;

SYMBOLDUMP :
PROCEDURE ;
DECLARE(LPM, I, J, K, L, M, PROCMARK) FIXED ;
DECLARE(BUFFER, BL) CHARACTER ;

```



```

DECLARE EXCHANGES BIT(1),SYTSORT(SYTLIMIT) BIT(16) ;

OUTLINE :
PROCEDURE(NAME,P) CHARACTER ;
DECLARE(NAME,TEMP) CHARACTER,(LN,P) FIXED ;
LN=LENGTH(NAME) ;
TEMP=NAME || SUBSTR(BLANKS,C,23-LN) ;
NAME=TYPE NAMES(SYTYPE(P)) ;
TEMP=TEMP || NAME || SUBSTR(BLANKS,0,10-LN) ;
NAME=SYTLOC(P) ;
LN=LENGTH(NAME) ;
TEMP=TEMP || AT ' ' NAME || SUBSTR(BLANKS,0,6-LN) ;
RETURN(TEMP) ;
TEMP=TEMP || DECLARED ON LINE ' ' DECLARED ON LINE(P)
TEMP=TEMP || REFERENCED ' ' REFER(P) || ' ' TIMES ' ' ;
END OUTLINE ;

PROC MARK = PROC MARK(DEPTH) ;
IF PROC MARK > SYTLN THEN OUTPUT = '*** SYMBOL TABLE EMPTY ***' ;
ELSE
DO ;
DOUBLE SPACE ;
SYMBOL TABLE DUMP : ' ' ;
OUTPUT=SYMBOL TABLE DUMP ;
DOUBLE SPACE ;
LPM=LENGTH(SYTLN) ;
L=15 ;
DO I = PROC MARK TO SYTLN ;
IF LENGTH(SYTLN(I)) > L THEN
L=LENGTH(SYTLN(I)) ;
END ;
IF L > 70 THEN L = 70 ;
SUBSTR(BLANKS,0,L) ;
DO I = PROC MARK TO SYTLN ;
SYTSORT(I)=I ;
K=LENGTH(SYTLN(I)) ;
IF K > 0 THEN IF K < L THEN
DO ;
BUFFER=SUBSTR(BL,K) ;
SYTLN(I)=SYTLN(I) || BUFFER ;
END ;
ELSE
DO ;
BUFFER=SUBSTR(SYTLN(I),0,L) ;
SYTLN(I)=BUFFER ;
END ;
EXCHANGES = TRUE ;
K=SYTLN - PROC MARK ;
DO WHILE EXCHANGES ;

```



```

EXCHANGES = FALSE ;
DO J = 0 TO K - 1 ;
  I = SYTLN - J ;
  L = I - 1 ;
  IF SYTID(SYTSORT(L)) > SYTID(SYTSORT(I)) THEN
    DO ;
      M = SYTSORT(I) ;
      SYTSORT(I) = SYTSORT(L) ;
      SYTSORT(L) = M ;
      EXCHANGES = TRUE ;
      K = J ; /* RECORD LAST SWAP */
    END ;
  END ;
END ;
PROC MARK ;
DO WHILE LENGTH(SYTID(SYTSORT(I))) = 0 ;
  I = I + 1 ;
END ;
DO I = I TO SYTLN ;
  K = SYTSORT(I) ;
  OUTPUT = OUTLINE(SYTID(K), K) ;
  K = K + 1 ;
  DO WHILE (LENGTH(SYTID(K)) = 0 ) & (K <= SYTLN) ;
    J = K - SYTSORT(I) ;
    OUTPUT = OUTLINE(, PARAMETER , || J || SUBSTR(BL, 14), K) ;
    K = K + 1 ;
  END ;
END ;
BUFFER = SUBSTR(SYTID(PROC MARK), 0, LPM) ;
SYTID(PROC MARK) = BUFFER ;
EJECT_PAGE ;
END ;
END SYMBOL DUMP ;

LOADER :
PROCEDURE ;
  /* BUILD ENTRY POINT */
  CALL FILE_36Q(BLANK || PROGRAM || FORMAT(0, BSS, ONE, 0)) ;
  /* BUILD ECS LOAD CALL TO MONITOR */
  LCNT = LCNT + 1 ;
  CALL EMIT_DATA(LCNT, VFD, VFD_START || ECSLOAD(0), 0) ;
  DO I = 1 TO RE_CNT ;

```



```

CALL EMIT_DATA(0,VFD,VFD_START || ECSLOAD(I),0) ;
END ;
CALL EMIT_COMPASS(0,61,2,0,0,LCNT) ;
CALL EMIT_COMPASS(0,61,3,0,0,-RE-CNT) ;
CALL EMIT_COMPASS(0,61,7,0,0,-7) ; /* LOADECS MONITOR CALL */
CALL EMITTO,'RJ',MONITOR,0) ;
CALL EMIT(C,EQ,BLANK,PROGRAM_START) ; /* BRANCH TO TOP OF CODE */
END LOADER ;

```

```

DUMPIT : PROCEDURE ;
CALL SYMBOL_DUMP ;
OUTPUT = ' MACRO ; DEFINITIONS : ' ;
DOUBLE_SPACE ;
DO I = 0 TO TOP_MACRO ;
OUTPUT = PAD(MACRO_NAME(I),20) || ' LITERALLY: ' || MACRO_TEXT(I) ;
END ;
DOUBLE_SPACE ;
/* PRINT OUT STATS */
OUTPUT = ' IDCOMPARES = ' || IDCOMPARES ;
OUTPUT = ' SYMBOL TABLE SIZE = ' || SYTLN ;
OUTPUT = ' MACRO DEFINITIONS = ' || TOP_MACRO + 1 ;
OUTPUT = ' STACKING DECISIONS = ' || CALLCOUNT(i) ;
OUTPUT = ' SCAN = ' || CALLCOUNT(3) ;
OUTPUT = ' FREE STRING AREA = ' || FREELIMIT - FREEBASE ;
OUTPUT = ' BLANK ; INSTRUCTION FREQUENCIES : ' ;
OUTPUT = ' BLANK ;
DO I = 0 TO 74 ;
IF INST_FREQ(I) = 0 THEN
OUTPUT = SUBSTR(CP_CODES,CP_MAP_2(I),2) || SUBSTR(BLANKS,0,4) ||
INST_FREQ(I) ;
END ;
DUMPIT ;

```

```

STACK_DUMP : PROCEDURE ;
DECLARE LINE CHARACTER ;
LINE = ' PARTIAL PARSE TO THIS POINT IS : ' ;
DO I = 2 TO SP ;
IF LENGTH(LINE) > 105 THEN
DO ;
OUTPUT = LINE ;
LINE = X4 ;
END ;

```



```

LINE = LINE || X1 || V(PARSE_STACK(I));
END;
OUTPUT = LINE;
END STACK_DUMP;

/*
THE SYNTHESIS ALGORITHM FOR XPL

SYNTHESIZE : PRODUCTION_NUMBER;
PROCEDURE( PRODUCTION_NUMBER FIXED ;
DECLARE PRODUCTION_NUMBER FIXED ;
IF CONTROL(BYTE('W')) THEN
    OUTPUT = ***; || PRODUCTION_NUMBER ;
DO CASE PRODUCTION_NUMBER ;
    ; /* NULL STMT */
/* <PROGRAM> ::= <STATEMENT LIST> */
DO ;
    COMPILING = FALSE ;
    CALL LOADER ;
    CALL EMIT(0, 'END', PROGRAM, 0) ;
END ;
/* <STATEMENT LIST> ::= <STATEMENT> */
; /* NULL STMT */
/* <STATEMENT LIST> ::= <STATEMENT LIST> <STATEMENT> */
; /* NULL STMT */
/* <STATEMENT> ::= <BASIC STATEMENT> */
; /* NULL STMT */
/* <STATEMENT> ::= <IF STATEMENT> */
; /* NULL STMT */
/* <BASIC STATEMENT> ::= <ASSIGNMENT> , • */
; /* NULL STMT */
/* <BASIC STATEMENT> ::= <GROUP> , • */
*/

```



```

; /* NULL STMT */

/* <BASIC STATEMENT> ::= <PROCEDURE> , . */

; /* NULL STMT */

/* <BASIC STATEMENT> ::= <GO TO STATEMENT> , . */

; /* NULL STMT */

/* <BASIC STATEMENT> ::= <CALL STATEMENT> , . */

; /* NULL STMT */

/* <BASIC STATEMENT> ::= <DECLARATION STATEMENT> , . */

; /* NULL STMT */

/* <BASIC STATEMENT> ::= <LABEL DEFINITION> <BASIC STATEMENT> */

; /* NULL STMT */

/* <BASIC STATEMENT> ::= <RETURN STATEMENT> */

; /* NULL STATEMENT */

/* <BASIC STATEMENT> ::= , . */

; /* NULL STMT */

/* <IF CLAUSE> ::= IF <EXPRESSION> THEN */

DO ;
J = FIXV(MPP1) ;
CALL EMIT_COMPASS(0,51,1,0,0,J) ; /* LOAD EXP VALUE */
LCNT = LCNT + 1 ;
CALL EMIT_COMPASS(0,3,0,1,0,LCNT) ; /* BRANCH OUT IF ZERO */
FIXV(MP) = LCNT ;
END ;

/* <TRUE PART> ::= <BASIC STATEMENT> ELSE */

DO ;
LCNT = LCNT + 1 ;
FIXV(MP) = LCNT ; /* LOCATION OF BRANCH OVER ELSE */
CALL EMIT(0,EQ,BLANK,LCNT) ;
LCNT = LCNT + 1 ;

```



```

    FIXL(MP) = LCNT ;
    CALL EMIT(LCNT,NO,BLANK,0) ;
    END ;

/* <IF STATEMENT> ::= <IF CLAUSE> <STATEMENT> */

    CALL EMIT(FIXV(MP),NO,BLANK,0) ;

/* <IF STATEMENT> ::= <IF CLAUSE> <TRUE PART> <STATEMENT> */

    DO ;
    J = FIXV(MPP1) ;
    CALL EMIT(0,EQ,BLANK,J) ;
    CALL EMIT(FIXV(MP),EQ,BLANK,FIXL(MPP1)) ;
    CALL EMIT(J,NO,BLANK,0) ;
    END ;

/* <IF STATEMENT> ::= <LABEL DEFINITION> <IF STATEMENT> */

    ; /* NULL STMT */

/* <ITERATION CONTROL> ::= TO <EXPRESSION> */

    DO ;
    FIXV(MP) = FIXV(MPP1) ; /* PASS LOCATION OF EXP */
    FIXL(MP) = EMIT_CONSTANT(1) ; /* RETURNS LOCATION OF CONSTANT 1 */
    END ;

/* <ITERATION CONTROL> ::= TO <EXPRESSION> BY <EXPRESSION> */

    DO ;
    FIXV(MP) = FIXV(MPP1) ; /* PASS LIMIT LOCATION */
    FIXL(MP) = FIXV(SP) ; /* PASS STEP LOCATION */
    END ;

/* <DUMMY> ::= <EXP> <IT CONT> */

    DO ;
    FIXL(MP) = FIXV(SP) ; /* PASS LIMIT */
    FIXO(MP) = FIXL(MPP1) ; /* PASS STEP LOCATION */
    END ;

/* <FOR HEAD> ::= DO <VARIABLE> <REPLACE> <DUMMY> */

    DO ; /* INITIALIZE DO VARIABLE */
    CALL EMIT_COMPASS(0,51,1,0,0,FIXV(MP+3)) ; /* LOAD INIT VALUE */
    CALL EMIT_COMPASS(0,10,7,1,0,0) ;

```



```

I = FIXV(MPP1) ;
CALL EMIT_COMPASS(0,51,7,0,0,I) ; /* STORE IN STEP VARIABLE */
LCNT = LCNT + 1 ;
LI = LCNT ;
CALL EMIT(0, EQ, BLANK, L1) ;
LCNT = LCNT + 1 ; /* STEP AND SAVE */
FIXV(MP) = LCNT ; /* PASS STEP CODE LOCATION */
CALL EMIT_COMPASS(LCNT, 51, 1, 0, 0, FIXV(MP+3)) ; /* <L2> SA1 <L> */
CALL EMIT_COMPASS(0, 51, 2, 0, 0, I) ; /* SA2 <I> */
CALL EMIT_COMPASS(0, 36, 7, 1, 2, 0) ; /* IX7 X1+X2 */
CALL EMIT_COMPASS(0, 51, 7, 0, 0, I) ; /* SA7 <I> */
CALL EMIT_COMPASS(0, 51, 7, 0, 0, I) ; /* TEST FOR LIMIT */
CALL EMIT_COMPASS(L1, 51, 2, 0, 0, FIXV(MP+3)) ; /* <L1> SA2 <K> */
CALL EMIT_COMPASS(0, 37, 3, 2, 1, 0) ; /* IX3 X2-X1 */
LCNT = LCNT + 1 ; /* PASS L3 */
FIXV(MP) = LCNT ;
CALL EMIT_COMPASS(0, 3, 2, 3, 0, LCNT) ; /* PL X3, <L3> */
END ;

/* <FOR HEAD> ::= <FOR HEAD> <STMT> */
; /* NULL STMT */

/* <DO WHILE> ::= DO WHILE */
DO ;
LCNT = LCNT + 1 ;
FIXV(MP) = LCNT ; /* PASS TOP OF DO WHILE STATEMENT */
CALL EMIT(LCNT, NO, BLANK, 0) ;
END ;

/* <WHILE HEAD> ::= <DO WHILE> <EXPRESSION> */
DO ;
LCNT = LCNT + 1 ;
FIXV(MP) = LCNT ; /* PASS BRANCH OUT OF DO WHILE STMT */
CALL EMIT_COMPASS(0, 51, 1, 0, 0, FIXV(MPP1)) ; /* SA1 <I> */
CALL EMIT_COMPASS(0, 3, 0, 1, 0, LCNT) ; /* ZR X1, <C> */
END ;

/* <WHILE HEAD> ::= <WHILE HEAD> <STATEMENT> */
; /* NULL STMT */

/* <CASE HEAD> ::= DO CASE <EXPRESSION> */
DO ;
CALL EMIT_COMPASS(0, 51, 1, 0, 0, FIXV(MP+2)) ; /* SA1 <A> */

```



```

CALL EMIT_COMPASS(0,63,2,1,0,0) ; /* SB2 X1 */
LCNT = LCNT + 1 ;
FIXV(MP) = LCNT ; /* PASS FIRST JUMP BACK LOC */
CALL EMIT_COMPASS(0,2,2,0,0,LCNT) ; /* JP B2+<B> */
LCNT = LCNT + 1 ;
FIXL(MP) = LCNT ; /* PASS BOTTOM OF CASE LOC */
LCNT = LCNT + 1 ;
FIXO(MP) = BCNT + 1 ; /* PASS CASE 0 MARKER */
BCNT = BCNT + 1 ;
BLOC(BCNT) = LCNT ; /* SAVE ACTUAL LOCATIONS */
CALL EMIT(LCNT,NO,BLANK,C) ;
END ;

/* <CASE HEAD> ::= <CASE HEAD> <STATEMENT> */

DO ;
CALL EMIT(0,EQ,BLANK,FXL(MP)) ; /* JUMP OUT LOC */
BCNT = BCNT + 1 ;
LCNT = LCNT + 1 ;
BLOC(BCNT) = LCNT ;
CALL EMIT(LCNT,NO,BLANK,C) ; /* TOP OF NEXT CASE */
INFORMATION = INFORMATION || 'CASE ' || BCNT-FIXO(MP) ;
END ;

/* <DO HEAD> ::= DO •, */
; /* NULL STMT */

/* <DO HEAD> ::= <DO HEAD> <STATEMENT> */
; /* NULL STMT */

/* <ENDING> ::= END */
; /* NULL STMT */

/* <ENDING> ::= END <IDENTIFIER> */
FIXV(MP) = SYTSEARCH(VAR(MP1)); /* FIND PROC SYT LOC */

/* <ENDING> ::= <LABEL DEFINITION> <ENDING> */
FIXV(MP) = FIXV(MP1) ; /* PASS PROC SYT LOC */

/* <GROUP> ::= <FOR HEAD><ENDING> */

```



```

DO ;
CALL EMIT(O,EQ,BLANK, FIXV(MP)) ; /* BUILD BRANCH BACK TO STEP */
CALL EMIT(FIXL(MP),NO,BLANK,O) ; /* BUILD BRANCH OUT */
END ;

/* <GROUP> ::= <WHILE HEAD> <ENDING> */

DO ;
CALL EMIT(O,EQ,BLANK, FIXV(MP)) ;
CALL EMIT(FIXL(MP),NO,BLANK,O) ;
END ;

/* <GROUP> ::= <CASE HEAD> <STATEMENT> <ENDING> */

DO ;
CALL EMIT(O,EQ,BLANK, FIXL(MP)) ;
N = FIXO(MP) ;
CALL EMIT(FIXV(MP),EQ,BLANK,BLOC(N)) ;
DO I = N TO BCNT ;
CALL EMIT(O,EQ,BLANK,BLOC(I)) ; /* BRANCH BACKS */
END ;
CALL EMIT(FIXL(MP),NO,BLANK,O) ;
BCNT = N ;
END ;

/* <GROUP> ::= <DO HEAD> <ENDING> */

; /* NULL STMT */

/* <PROCEDURE NAME> ::= <IDENTIFIER> . . PROCEDURE */
DO ;
/* BUILD A BRANCH AROUND THE PROCEEDURE */
LCNT = LCNT + 1 ;
CALL EMIT(O,EQ,BLANK,LCNT) ;
LCNT = LCNT + 1 ;
FIXV(MP) = ENTER3(VAR(MP),LCNT,PROC) ;
CALL EMIT(LCNT,BSS,ONE,O) ;
END ;

/* <PROCEDURE HEAD> ::= <PROCEDURE NAME> , . */

DO ;
I = FIXV(MP) ;
SYTTYPE(I) = PROC ;
END ;

```



```

/* ALL PROCEDURES ARE CONSTRUCTED IN THE FOLLOWING MANNER :

    <ENTRY LABEL>
    EQ
    NO
    <LABEL OVER>
    EQ
    NO
    <RETURN VALUE>
    BSS 1
    <PARAMETER 1>
    BSS 1

    <LAST PARAMETER>
    BSS 1
    <LABEL 1>
    NO
    <CODE FOR PROCEDURE >
    NO
    <LABEL OVER>
    NO
    */

/* <PROCEDURE HEAD> ::= <PROCEDURE NAME> <TYPE> , • */

DO ;
J = FIXV(MP) ;
LCNT = LCNT + 1 ;
I = LCNT ;
CALL EMIT(O, EQ, BLANK, I) ;
LCNT = LCNT + 1 ;
CALL EMIT(LCNT, BSS, ONE, O) ;
CALL EMIT(I, NO, BLANK, O) ;
SYTTYPE(J) = FIXV(MPPI) + PROC ; /* SET TYPE */
END ;

/* <PROCEDURE HEAD> ::= <PROCEDURE PARAM LIST> , • */

CALL EMIT(FIXO(MP), NO, BLANK, O) ;

/* <PROCEDURE HEAD> ::= <PROCEDURE PARAM LIST> <TYPE> , • */

DO ;
CALL EMIT(FIXO(MP), NO, BLANK, O) ;
SYTTYPE(FIXV(MP)) = FIXV(MP+2) + PROC ;
END ;

/* <PROCEDURE PARAM LIST> ::= <PROCEDURE NAME> ( */

PRAMCNT(FIXV(MP)) = 0 ; /* SET PRAMETER COUNT */

/* <PROCEDURE PARAM LIST> ::= <PROCEDURE PARAM LIST> <IDENTIFIER> */

DO ;
LCNT = LCNT + 1 ;
FIXO(MP) = LCNT ; /* SAVE TOP OF BRANCH OVER PRAMETERS */
CALL EMIT(O, EQ, BLANK, LCNT) ;

```



```

LCNT = LCNT + 1 ;
CALL EMIT(LCNT,BSS,ONE,0) ;
LCNT = LCNT + 1 ;
I = ENTER3(VAR(MPP1),LCNT,PARAM) ;
CALL EMIT(LCNT,BSS,ONE,0) ;
PRAMCNT(FIXV(MP)) = PRAMCNT(FIXV(MP)) + 1 ; /* STEP PRAM CNT */
END ;

/* <PROCEDURE PARAM LIST> ::= <PROCEDURE PARAM LIST> , <IDENTIFIER> */

DO ;
LCNT = LCNT + 1 ;
I = ENTER3(VAR(MP+2),LCNT,PARAM) ;
CALL EMIT(LCNT,BSS,ONE,0) ;
PRAMCNT(FIXV(MP)) = PRAMCNT(FIXV(MP)) + 1 ; /* STEP PRAM CNT */
END ;

/* PROCEDURE DEFINITION ::= <PROCEDURE HEAD> <STATEMENT> */
; /* NULL STMT */

/* <PROCEDURE DEFINITION> ::= <PROCEDURE DEFINITION> <STATEMENT> */

/* <PROCEDURE> ::= <PROCEDURE DEFINITION> <ENDING> */
; /* NULL STMT */

DO ;
I = FIXV(MP) ;
IF I = FIXV(MPP1) THEN
CALL ERROR('CHECK PROC ENDING IDENTIFIER',1) ;
CALL EMIT(0,EQ,BLANK,SYTLOC(I)) ;
CALL EMIT(SYTLOC(I)-1,NO,BLANK,0) ; /* FINISH BRANCH OVER PROC */
CALL CUTBACK ;
END ;

/* <RETURN STATEMENT> ::= RETURN */

DO ;
I = SYTLOC(PROC_MARK(DEPTH)) ; /* RETURN JUMP LOCATION */
CALL EMIT(0,EQ,BLANK,I) ;
END ;

/* <RETURN STATEMENT> ::= RETURN <EXPRESSION> */

DO ;
I = PROC_MARK(DEPTH) ;
IF SYTTYPE(I) = FIXEDPROC THEN

```



```

END ;

/* <SUBSCRIPT HEAD> ::= <SUBSCRIPT HEAD> <EXPRESSION> , */

DO ; TYPE(MP) ;
IF K = PROC | K = FIXEDPROC | K = CHRPROC THEN
DO ; PROC CALLS 3 */
CALL EMIT_COMPASS(0,51,1,0,C, FIXV(MPPI)); /* SA1 <I> */
CALL EMIT_COMPASS(0,10,7,1,0,0); /* BX7 X1 */
FIXL(MP) = FIXL(MP) + 1; /* STEP M */
I = SYTLOC(FIXV(MP) + FIXL(MP));
CALL EMIT_COMPASS(0,51,7,0,C,I); /* SA7 <B> */
END ;
ELSE
CALL ERROR('MULTIPLE SUBSCRIPTS ARE ILLEGAL',1) ;
END ;

/* <VARIABLE> ::= <IDENTIFIER> */

DO ; SYTSEARCH(VAR(MP)) ;
IF I < 0 | SYTYPE(I) = PARAM THEN
CALL ERROR('UNDECLARED VARIABLE ' || VAR(MP) , 1) ;
FIXV(MP) = I; /* PASS N */
TYPE(MP) = SYTYPE(I) ;
IF TYPE(MP) = FIXEDTYPE THEN
DO ;
FIXL(MP) = 0 ; /* INDICATES THIS IS A DIRECT REFERENCE */
FIXV(MP) = SYTLOC(I); /* PASS SYMBOL TABLE LOC */
FIXO(MP) = ORIGINAL; /* INDICATES PROTECTION MIGHT BE NEEDED */
END ;
END ;

/* <VARIABLE> ::= <SUBSCRIPT HEAD> <EXPRESSION> ) */
/* <I> */

DO ; FIXV(MPPI) ;
CALL EMIT_COMPASS(0,51,1,0,0,I); /* SA1 <I> */
K = TYPE(MP) ;
N = FIXV(MP) ; /* SYT LOCATION OF PROC ENTRY */
L = FIXL(MP) ; /* DISPLACEMENT IN SYT FROM PROC */
IF K = PROC | K = FIXEDPROC | K = CHRPROC THEN
DO ;
CALL EMIT_COMPASS(0,10,7,1,0,C); /* BX7 X1 */

```



```

L = L + 1;
J = SYTLOC(N+L); /* ACTUAL LOCATION OF PRAMETER */
CALL EMIT_COMPASS(0,51,7,0,0,J); /* SA7 <B> */
IF L < PRAMCNT(N) THEN
DO; /* MUST ZERO OUT REMAINING PRAMETERS */
IF N = MONITOR CALL THEN
CALL ERROR('TOO FEW PARAMETERS',0);
CALL EMIT(C,'SX7',BC+8C,0); /* MUST STEP AROUND BUG */
CALL DO WHILE L <= PRAMCNT(I);
J = L+1;
J = SYTLOC(N+L);
CALL EMIT_COMPASS(0,51,7,0,0,J);
END;
END; /* PROTECTION MAY BE NEEDED */
FIXO(MP) = ORIGINAL; /* PROTECTION MAY BE NEEDED */
END;
ELSE
/* K = FIXEDTYPE | K = CHRTYPE */
DO;
J = SYTLOC(FIXV(MP));
CALL EMIT_COMPASS(0,72,7,1,0,J); /* SX7 X1+<A> */
I = PROTECT(MPPI); /* MUST INDICATE AN INDIRECT REFERENCE */
FIXL(MP) = -1;
FIXV(MP) = I;
END;
END;

/* <PRIMARY> ::= <CONSTANT> */
FIXO(MP) = ORIGINAL; /* PROTECTION MAY BE NEEDED */

/* <PRIMARY> ::= <VARIABLE> */
DO;
K = TYPE(MP);
IF K = FIXEDPROC | K = CHRPROC THEN
DO;
I = SYTLOC(FIXV(MP));
CALL EMIT_COMPASS(0,1,0,0,0,I); /* RJ <PROC> */
CALL EMIT_COMPASS(0,51,1,0,0,I+2); /* SA1 B2+<PROC+2> */
CALL EMIT_COMPASS(0,10,7,1,0,0); /* BX7 X1 */
FIXV(MP) = PROTECT(MP);
END;
ELSE
IF K = PROC & FIXL(MP) < 0 THEN
/* AN INDIRECT REFERENCE, MUST LOAD AGAIN */
DO;
CALL EMIT_COMPASS(0,51,1,0,0,FIXV(MP)); /* LOAD LOCATION */

```



```

CALL EMIT_COMPASS(0,53,1,1,0,0); /* SAL X1+BQ */
CALL EMIT_COMPASS(0,10,7,1,0,0); /* BX7 X1 */
FIXV(MP) = PROTECT(MP);
END;
/* CHECK AND RESET VARIABLE TYPES */
IF K = FIXEDPROC THEN K = FIXEDTYPE;
ELSE IF K = CHRPROC THEN K = CHRTYPE;
ELSE IF K = PROC THEN
CALL ERROR(SYUID(I)) || 'DOES NOT RETURN A VALUE',1);
/* ONLY TWO TYPES LEFT NOW, FIXEDTYPE AND CHRTYPE */
END;

/* <PRIMARY> ::= ( <EXPRESSION> ) */
DO;
TYPE(MP) = TYPE(MPPI);
FIXV(MP) = FIXV(MPPI); /* PASS LOCATION */
FIXO(MP) = FIXO(MPPI); /* PASS LOCATION TYPE */
END;

/* <TERM> ::= <PRIMARY> */
; /* NULL STMT */

/* <TERM> ::= <TERM> * <PRIMARY> */
CALL FLCAT_OPS(40);

/* <TERM> ::= <TERM> / <PRIMARY> */
CALL FLOAT_OPS(44);

/* <ARITHMETIC_EXPRESSION> ::= <TERM> */
; /* NULL STMT */

/* <ARITHMETIC_EXPRESSION> ::= <ARITHMETIC_EXPRESSION> + <TERM> */
CALL FIXED_OPS(36);

/* <ARITHMETIC_EXPRESSION> ::= <ARITHMETIC_EXPRESSION> - <TERM> */
CALL FIXED_OPS(37);

/* <ARITHMETIC_EXPRESSION> ::= + <TERM> */
DO;
FIXV(MP) = FIXV(MPPI);
TYPE(MP) = TYPE(MPPI);

```



```

      FIXO(MP) = FIXO(MPP1) ;
END ;

/* <ARITHMETIC EXPRESSION> ::= - <TERM> */
/* <I> */
DO ;
CALL EMIT_COMPASS(0,71,1,0,0,0) ; /* SX1 0 */
I = FIXV(MPP1) ; /* <I> */
CALL EMIT_COMPASS(0,51,2,0,0,1) ; /* SA2 IX7 X1-X2 */
CALL EMIT_COMPASS(0,37,7,1,2,0) ; /* */
FIXV(MP) = PROTECT(MPP1) ;
TYPE(MP) = TYPE(MPP1) ;
END ;

/* <STRING EXPRESSION> ::= <ARITHMETIC EXPRESSION> */
; /* NULL STMT */

/* <STRING EXPRESSION> ::= <STRING EXPRESSION> CAT <ARITHMETIC EXPRESSION> */
/* */

; /* NULL STMT */

/* <RELATION> ::= EQ */
FIXV(MP) = 0 ;

/* <RELATION> ::= LT */
FIXV(MP) = 2 ;

/* <RELATION> ::= GT */
FIXV(MP) = 1 ;

/* <RELATION> ::= NE */
FIXV(MP) = 5 ;

/* <RELATION> ::= NL */
FIXV(MP) = 3 ;

/* <RELATION> ::= NG */
FIXV(MP) = 4 ;

```



```

/* <RELATION> ::= LE */
    FIXV(MP) = 4 ;
/* <RELATION> ::= GE */
    FIXV(MP) = 3 ;
/* <LOGICAL PRIMARY> ::= <STRING EXPRESSION> */
    ; /* NULL STMT */
/* <LOGICAL PRIMARY> ::= <STRING EXPRESSION> <RELATION> <STRING EXPRESSION>
*/
    IF TYPE(MP) = CHRTYPE & TYPE(SP) = CHRTYPE THEN
/* TWO FIXED VARIABLES IN LOGICAL EXPRESSION */
    DO ;
        CALL EMIT_COMPASS(0,51,1,0,0, FIXV(MP+2)); /* SAI <J> */
        I = FIXV(MP) ;
        CALL EMIT_COMPASS(0,51,2,0,0,1); /* SA2 <I> */
        CALL EMIT_COMPASS(3,37,3,2,1,0); /* IX3 X2-X1 */
        CALL EMIT_COMPASS(0,71,7,0,0,-1); /* SX7 1 */
        LCNT = LCNT + 1 ;
    DO CASE FIXV(MP+1) ; /* N */
/* 0 */
        J = 0 ; /* ZR X3,<K> */
/* 1 */
        J = 1 ; /* PL X3,<K> */
/* 2 */
        J = 2 ; /* NG X3,<K> */
/* 3 */
        J = 3 ; /* NG X3,<K> */
    DO ;
        CALL EMIT_COMPASS(0,3,2,3,0,LCNT); /* PL X3,<K> */
        J = 0 ;
    END ;
/* 4 */
    DO ;
        CALL EMIT_COMPASS(0,3,3,3,0,LCNT); /* NG X3,<K> */
        J = 0 ;
    END ;
/* 5 */
        J = 1 ; /* NZ X3,<K> */
    END ; /* OF CASE */
    CALL EMIT_COMPASS(0,3,J,3,0,LCNT); /* SEE ABOVE CASES */
/* BUG IN EMIT_COMPASS */
    CALL EMIT(0,'SX7','BO+BQ',0) ;

```



```

CALL EMIT(LCNT,NO,BLANK,0);
FIXV(MP) = PROTECT(MP) ;
END ;
ELSE
DO ;
IF TYPE(MP) = CHRTYPE & TYPE(SP) = CHRTYPE THEN
DO ;
IF FIXV(MPP1) = 0 THEN CALL ERROR('ILLEGAL STRING RELATION',1);
ELSE CALL STRING_ADDITION(1);
END ;
ELSE
CALL ERROR('MIXED TYPES IN LOGICAL EXPRESSION',1);
END ;

/* <LOGICAL SECCNDARY> ::= <LOGICAL PRIMARY> */

/* <LOGICAL SECONDARY> ::= NOT <LOGICAL PRIMARY> */

DO ;
I = FIXV(MPP1) ;
TYPE(MP) = TYPE(MPP1) ; /* PASS TYPE */
CALL EMIT_COMPASS(0,51,1,0,C,I) ; /* SA1 <I> */
CALL EMIT_COMPASS(0,71,7,0,1,C) ; /* SX7 1 */
LCNT = LCNT + 1 ; /* SX7 1 */
CALL EMIT_COMPASS(0,3,C,1,0,LCNT) ; /* ZR X1,<K> */
CALL EMIT_COMPASS(0,71,7,0,0,0) ; /* SX7 0 */
CALL EMIT(LCNT,NO,BLANK,0) ;
FIXV(MP) = PROTECT(MPP1) ;
END ;

/* <LOGICAL FACTOR> ::= <LOGICAL SECONDARY> */

/* <LOGICAL FACTOR> ::= <LOGICAL FACTOR> AND <LOGICAL SECONDARY> */

CALL LOGIC_OPS(11) ;

/* <EXPRESSION> ::= <LOGICAL FACTOR> */

/* <EXPRESSION> ::= <EXPRESSION> OR <LOGICAL FACTOR> */

CALL LOGIC_OPS(12) ;

```



```

/* <IDENTIFIER LIST> ::= (      */
    FIXV(MP) = BCNT ; /* PASS SEPARATE ID LOC STACK PTR */
/* <IDENTIFIER LIST> ::= <IDENTIFIER LIST> <IDENTIFIER> , */
    DO ;
    /* STORE SYT LOC OF ID IN SEPARATE STACK */
    BLOC(BCNT) = ENTER3(VAR(MPPI),0,0);
    BCNT = BCNT + 1;
    END ;

/* <IDENTIFIER SPECIFICATION> ::= <IDENTIFIER> */
    DO ;
    FIXV(MP) = BCNT ;
    BLOC(BCNT) = ENTER3(VAR(MP),0,0) ; /* PASS ID LOC STACK PTR */
    END ;

/* <IDENTIFIER SPECIFICATION> ::= <IDENTIFIER LIST> <IDENTIFIER> ) */
    BLOC(BCNT) = ENTER3(VAR(MPPI),0,0) ;

/* <BOUND HEAD> ::= <IDENTIFIER SPECIFICATION> (      */
    ; /* NULL STMT */

/* <BOUND HEAD> ::= <BOUND HEAD> <NUMBER> ) */
    FIXL(MP) = FIXV(MPPI) ; /* PASS ARRAY LENGTH */

/* <TYPE DECLARATION> ::= <IDENTIFIER SPECIFICATION> <TYPE> */
    DO ;
    FIXO(MP) = FIXV(MPPI) ; /* PASS TYPE */
    FIXL(MP) = 1 ; /* PASS ARRAY LN */
    END ;

/* <TYPE DECLARATION> ::= <BOUND HEAD> <TYPE> */
    FIXO(MP) = FIXV(MPPI) ; /* PASS TYPE */

/* <TYPE DECLARATION> ::= <INITIAL HEAD> ) */
    DO ;
    IF FIXL(MP) > 0 THEN
        CALL ERROR('TOO FEW INITIAL ENTRIES',1) ;
    IF FIXL(MP) < 0 THEN

```



```

CALL ERROR('TOO MANY INITIAL ENTRIES',1) ;
FIXL(MP) = 0 ;
END ;

/* <INITIAL> ::= INITIAL ( /*/
; /* NULL STMT */

/* <INITIAL HEAD> ::= <TYPE DECLARATION> <INITIAL> <NUMBER> */

IF SYTTYPE(BLOC(FIXV(MP))) = PARAM THEN
CALL ERROR('ILLEGAL TO INITIALIZE A PROC PARAMETER',1) ;
ELSE
DO ;
FIXL(MP) = FIXL(MP) - 1 ; /* DECREMENT Y */
LCNT = LCNT + 1 ;
J = FIXV(MP) ;
SYTLOC(BLOC(J)) = LCNT ;
SYTTYPE(BLOC(J)) = FIXO(MP) ; /* SET TYPE */
IF J = BCNT THEN
CALL ERROR('ILLEGAL TO INITIALIZE AN IDENTIFIER LIST',1) ;
ELSE
DO ;
FIELD = VFD START ;
FIELD = FIELD || FIXV(MP+2) ;
CALL EMIT_DATA(LCNT,VFD,FIELD,0) ;
END ;
END ;

/* <INITIAL HEAD> ::= <INITIAL HEAD> , <NUMBER> */

DO ;
FIELD = VFD START ;
FIELD = FIELD || FIXV(MP+2) ;
CALL EMIT_DATA(0,VFD,FIELD,C) ;
FIXL(MP) = FIXL(MP) - 1 ;
END ;

/* <INITIAL HEAD> ::= <TYPE DECLARATION> <INITIAL> <STRING> */

; /* NULL STATEMENT */

/* <INITIAL HEAD> ::= <INITIAL HEAD> , <STRING> */

; /* NULL STATEMENT */

/* <BIT HEAD> ::= BIT ( */

```



```

; /* NULL STMT */

/* <TYPE> ::= FIXED */
    FIXV(MP) = FIXEDTYPE ;

/* <TYPE> ::= CHARACTER */
    FIXV(MP) = CHRTYPE ;

/* <TYPE> ::= LABEL */
    FIXV(MP) = LABELTYPE ;

/* <TYPE> ::= <BIT HEAD> <NUMBER> ) */
    FIXV(MP) = FIXEDTYPE ;

/* <DECLARATION ELEMENT> ::= <TYPE DECLARATION> */
DO ;
K = FIXO(MP) ; /* VARIABLE TYPE */
I = FIXL(MP) ;
N = FIXV(MP) ; /* BCNT */
IF I THEN
DO J = N TO BCNT ;
L = SYTYPE(BLOC(J)) ;
SYTYPE(BLOC(J)) = K ;
IF L THEN PARAM THEN
DO ;
LCNT = LCNT + 1 ;
FIELD = I ;
CALL EMIT_DATA(LCNT,BSS,FIELD ,0) ; /* <I> BSS K */
SYTLOC(BLOC(J)) = LCNT ;
END ;
END ;
BCNT = N ; /* RESET SEPARATE STACK COUNTER */
END ;

/* <DECLARATION ELEMENT> ::= <IDENTIFIER> LITERALLY <STRING> */
IF TOP_MACRO > = MACRO_LIMIT THEN
CALL ERROR(' MACRO TABLE OVERFLOW',1) ;
ELSE
DO ;
TOP_MACRO = TOP_MACRO + 1 ;
I = LENGTH(VAR(MP)) ;
J = MACRO_INDEX(I) ;

```



```

DO L = 1 TO TOP_MACRO - J ;
K = TOP_MACRO - L ;
MACRO_NAME(K+1) = MACRO_NAME(K) ;
MACRO_TEXT(K+1) = MACRO_TEXT(K) ;
END ;
MACRO_NAME(J) = VAR(SP) ;
MACRO_TEXT(J) = VAR(SP) ;
DO J = 1 TO 225 ;
MACRO_INDEX(J) = MACRO_INDEX(J) + 1 ;
END ;

END ;

/* <DECLARATION STATEMENT> ::= DECLARE <DECLARATION ELEMENT> */

; /* NULL STMT */

/* <DECLARATION STATEMENT> ::= <DECLARATION STATEMENT> , <DECLARATION ELEMENT> */
; /* NULL STMT */

/* <GO TO> ::= GO TO */
; /* NULL STMT */

/* <GO TO> ::= GOTO */
; /* NULL STMT */

/* <GO TO STATEMENT> ::= <GO TO> <IDENTIFIER> */

DO ; SYTSEARCH(VAR(MP)) ;
IF I < 0 THEN
DO ;
LCNT = LCNT + 1 ;
CALL ENTER1(VAR(MP),LCNT,LABELTYPE) ;
I = LCNT ;
END ;
IF SYTTYPE(I) = LABELTYPE THEN
CALL ERROR('GO TO A NON-LABEL TYPE IDENTIFIER',1) ;
ELSE
CALL EMIT_COMPASS(0,2,0,0,I) ;
END ;

/* <REPLACE> ::= = */
; /* NULL STMT */

```



```

/* <LEFT PART> ::= <VARIABLE> , */
; /* NULL STMT */

/* <ASSIGNMENT> ::= <VARIABLE> <REPLACE> <EXPRESSION> */
CALL ASSIGN ;

/* <ASSIGNMENT> ::= <LEFT PART> <ASSIGNMENT> */
CALL ASSIGN ;

/* <LABEL DEFINITION> ::= <IDENTIFIER> . . */

DO ; SYTSEARCH(VAR(MP)) ;
IF I < G THEN
DO ;
LCNT = LCNT + 1 ;
CALL ENTER1(VAR(MP),LCNT,LABELTYPE) ;
I = LCNT ;
END ;
CALL EMIT(I,NO,BLANK,0) ;
END ;

END ; /* SYNTHESIZE CASE STMT */

END SYNTHESIZE;

/* SYNTACTIC PARSING FUNCTIONS */

RIGHT_CONFLICT:
PROCEDURE (LEFT) BIT(1);
DECLARE LEFT FIXED;
/* THIS PROCEDURE IS TRUE IF TOKEN IS A LEGAL RIGHT CONTEXT OF LEFT*/
RETURN ("CO" & SHL(BYTE(CI(LEFT)), SHR(TOKEN,2)), SHL(TOKEN,1)
& "O6")) = 0;
END RIGHT_CONFLICT;

RECOVER:

```



```

PROCEDURE; THIS IS THE SECOND SUCCESSIVE CALL TO RECOVER, DISCARD ONE SYMBOL */
/* IF FAILSOFT THEN CALL SCAN;
FAILSOFT = FALSE;
DO WHILE ~ STOPIT(TOKEN);
CALL SCAN; /* TO FIND SOMETHING SOLID IN THE TEXT */
END;
DC WHILE RIGHT CONFLICT (PARSE_STACK(SP));
IF SP > 2 THEN SP = SP - 1; /* AND IN THE STACK */
ELSE CALL SCAN; /* BUT DON'T GO TOO FAR */
END;
OUTPUT = 'RESUME:' || SUBSTR(POINTER, TEXT_LIMIT-CP+MARGIN_CHOP+7);
END RECOVER;

STACKING:
PROCEDURE BIT(1); /* STACKING DECISION FUNCTION */
CALLCOUNT(1) = CALLCOUNT(1) + 1;
DO FOREVER; /* UNTIL RETURN */
DO CASE SHR(BYTE(CI(PARSE_STACK(SP)), SHR(TOKEN, 2)), SHL(3-TOKEN, 1)&6)&3;
/* CASE 0 */
DO; /* ILLEGAL SYMBOL PAIR */
CALL ERROR('ILLEGAL SYMBOL PAIR: ' || V(PARSE_STACK(SP)) || X1 ||
V(TOKEN), 1);
CALL STACK_DUMP;
CALL RECOVER;
END;
/* CASE 1 */
RETURN TRUE; /* STACK TOKEN */
/* CASE 2 */
RETURN FALSE; /* DON'T STACK IT YET */
/* CASE 3 */
DO; /* MUST CHECK TRIPLES */
J = SHL(PARSE_STACK(SP-1), 16) + SHL(PARSE_STACK(SP), 8) + TOKEN;
I = -1; K = NCITRIPLES + 1; /* BINARY SEARCH OF TRIPLES */
DO WHILE I + 1 < K;
L = SHR(I+K, 1);
IF CITRIPLES(L) > J THEN K = L;
ELSE IF CITRIPLES(L) < J THEN I = L;
ELSE RETURN TRUE; /* IT IS A VALID TRIPLE */
END;
RETURN FALSE;

```



```

END;
END; /* OF DO CASE */
END; /* CF DO FOREVER */
END STACKING;

PR_OK: PROCEDURE (PRD) BIT(1);
/* DECISION PROCEDURE FOR CONTEXT CHECK OF EQUAL OR IMBEDDED RIGHT PARTS */
DECLARE (H, I, J, PRD) FIXED;
DO CASE CONTEXT_CASE (PRD);
/* CASE 0 -- NO CHECK REQUIRED */
RETURN TRUE;
/* CASE 1 -- RIGHT CONTEXT CHECK */
RETURN ~ RIGHT_CONFLICT (HDTB (PRD));
/* CASE 2 -- LEFT CONTEXT CHECK */
DO; H = HDTB (PRD) - NT;
I = PARSE_STACK (SP - PRLENGTH (PRD));
DO J = LEFT_INDEX (H-1) TO LEFT_INDEX (H) - 1;
IF LEFT_CONTEXT (J) = I THEN RETURN TRUE;
END;
RETURN FALSE;
END;
/* CASE 3 -- CHECK TRIPLES */
DO; H = HDTB (PRD) - NT;
I = SHL (PARSE_STACK (SP - PRLENGTH (PRD)), 8) + TOKEN;
DO J = TRIPLE_INDEX (H-1) TO TRIPLE_INDEX (H) - 1;
IF CONTEXT_TRIPLE (J) = I THEN RETURN TRUE;
END;
RETURN FALSE;
END;
END; /* OF DO CASE */
END PR_OK;
/*
ANALYSIS ALGORITHM
*/

```



```

REDUCE:
PROCEDURE;
  DECLARE (I, J, PRD) FIXED;
  /* PACK STACK TOP INTO ONE WORD */
  DO I = SP - 4 TO SP - 1;
  DO J = SHL(J, 8) + PARSE_STACK(I);
  END;

  DO PRD = PR_INDEX(PARSE_STACK(SP)-1) TO PR_INDEX(PARSE_STACK(SP)) - 1;
  IF (PRMASK(PLENGTH(PRD)) & J) = PRTB(PRD) THEN
    DO;
      7* AN ALLOWED REDUCTION */
      MP = SP - PLENGTH(PRD) + 1; MPPI = MP + 1;
      CALL SYNTHESIZE(PRTB(PRD));
      SP = MP;
      PARSE_STACK(SP) = HDTB(PRD);
      RETURN;
    END;

  END;

  /* LOOK UP HAS FAILED, ERROR CONDITION */
  CALL ERROR('NO PRODUCTION IS APPLICABLE', 1);
  CALL STACK_DUMP;
  FAILSOFT = FALSE;
  CALL RECOVER;
  END REDUCE;

COMPILE_LOOP:
PROCEDURE;
  CCMPILING = TRUE;
  DO WHILE CCMPILING;
    DO WHILE STACKING;
      SP = SP + 1;
      IF SP = STACKSIZE THEN
        DO;
          CALL ERROR('STACK OVERFLOW *** CHECKING ABORTED ***', 2);
          RETURN; /* THUS ABORTING CHECKING */
        END;
      PARSE_STACK(SP) = TOKEN;
      VAR(SP) = BCD;
      FIXV(SP) = NUMBER_VALUE;
      LAST_TOKEN = LAST_TOKEN;
      LAST_TOKEN = TOKEN;
      CALL SCAN;
    END;
  END;

```



```

        CALL REDUCE; DO WHILE COMPILING */
        END;
        END COMPILATION_LOOP;

PRINT_SUMMARY:
PROCEDURE;
DECLARE I FIXED;
CALL PRINT_DATE_AND_TIME ('END OF CHECKING ', DATE, TIME);
OUTPUT = '';
OUTPUT = CARD_COUNT || ' CARDS WERE CHECKED.';
IF ERROR_COUNT = 0 THEN OUTPUT = 'NO ERRORS WERE DETECTED.';
ELSE IF ERROR_COUNT > 1 THEN
    OUTPUT = 'SEVERE ERRORS WERE DETECTED.';
    IF SEVERE_ERRORS = 1 THEN OUTPUT = 'ONE SEVERE ERROR WAS DETECTED.';
    IF PREVIOUS_ERROR = 0 THEN
        OUTPUT = 'THE LAST DETECTED ERROR WAS ON LINE ' || PREVIOUS_ERROR
        || PERIOD;
    IF CONTROL(BYTE('D')) THEN CALL DUMPT;
    DOUBLE SPACE;
    CLOCK(3) = TIME;
    DO I = 1 TO 3;
        IF CLOCK(I) < CLOCK(I-1) THEN CLOCK(I) = CLOCK(I) + 8640000;
    END;
    CALL PRINT_TIME ('TOTAL TIME IN CHECKER ', CLOCK(3) - CLOCK(0));
    CALL PRINT_TIME ('SET UP TIME ', CLOCK(1) - CLOCK(0));
    CALL PRINT_TIME ('ACTUAL CHECKING TIME ', CLOCK(2) - CLOCK(1));
    CALL PRINT_TIME ('CLEAN-UP TIME AT END ', CLOCK(3) - CLOCK(2));
    IF CLOCK(2) > CLOCK(1) THEN /* WATCH OUT FOR CLOCK BEING OFF */
        OUTPUT = 'CHECKING RATE: ' || 6000*(CARD_COUNT/(CLOCK(2)-CLOCK(1)))
        || ' CARDS PER MINUTE.';
    END PRINT_SUMMARY;

MAIN_PROCEDURE:
PROCEDURE;
CLOCK(0) = TIME; /* KEEP TRACK OF TIME IN EXECUTION */
CALL INITIALIZATION;

CLOCK(1) = TIME;

CALL COMPILATION_LOOP;

CLOCK(2) = TIME;

```



```
/* CLOCK(3) GETS SET IN PRINT_SUMMARY */  
CALL PRINT_SUMMARY;  
END MAIN_PROCEDURE;  
  
CALL MAIN_PROCEDURE;  
RETURN SEVERE_ERRORS;  
EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF EOF
```


LIST OF REFERENCES

1. Control Data Technical Publication No. 60100000, Control Data 6400/6500/6600 Computer Systems Reference Manual, page 1-1, 1969.
2. Davis, G. B., IBM System/360 Computer, Chapter 2, McGraw-Hill, 1965.
3. Earley, J., and Sturgis, H., "A Formalism for Translator Interactions", CACM, v. 13-10, pages 607-617, October 1970.
4. Halstead, M. H., Machine-Independent Computer Programming, Spartan, 1962.
5. IBM Systems Reference Library No. S360-29, IBM System/360 PL/I Reference Manual, p. 200, March 1968.
6. IBM Systems Reference Library No. 6N22-6821-7, IBM System/360 Principles of Operation, p. 5, Nov. 1970.
7. McKeeman, W. M., Horning, J. J., and Wortman, D. B., A Compiler Generator, Prentice-Hall, 1970.
8. McKeeman, W. M., and others, "The XPL Compiler Generator System*", FJCC v. , p. 619-635, 1968.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2. Library, Code 0202 Naval Postgraduate School Monterey, California 93940	2
3. LTJG G. A. Kildall, Code 53Kd Department of Mathematics Naval Postgraduate School Monterey, California 93940	2
4. ENS Ronald Crocker Smeder, USNR 1550 Zuleta Avenue Coral Gables, Florida 33146	2

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Naval Postgraduate School Monterey, California 93940		2a. REPORT SECURITY CLASSIFICATION Unclassified	
		2b. GROUP	
3. REPORT TITLE An Investigation of the Bootstrapping Process as Applied to Compiler Generation			
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) Master's Thesis; June 1971			
5. AUTHOR(S) (First name, middle initial, last name) Ronald C. Smeder			
6. REPORT DATE June 1971	7a. TOTAL NO. OF PAGES 165	7b. NO. OF REFS 8	
8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)		
b. PROJECT NO.			
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)		
d.			
10. DISTRIBUTION STATEMENT Approved for public release; distribution unlimited.			
11. SUPPLEMENTARY NOTES		12. SPONSORING MILITARY ACTIVITY Naval Postgraduate School Monterey, California 93940	
13. ABSTRACT This paper examines the problem of compiler generation using bootstrapping techniques. It summarizes the development of several compiler systems which were produced through bootstrapping and presents the methods and formalisms through which they can be examined. The discussion of these techniques is illustrated with the development of an XPL compiler system for the Control Data 6500 computer.			

14

KEY WORDS

LINK A

LINK B

LINK C

ROLE

WT

ROLE

WT

ROLE

WT

Bootstrapping

Compiler generation

XPL

128146

Thesis
S57193
c.1

Smeder

An investigation of
the bootstrapping
process as applied to
compiler generation

17 OCT 78
17 OCT 84

2607
29820

Thesis
S57193
c.1

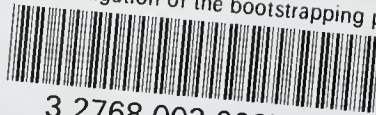
Smeder

An investigation of
the bootstrapping
process as applied to
compiler generation.

128146

thesS57193

An investigation of the bootstrapping pr



3 2768 002 00870 8

DUDLEY KNOX LIBRARY